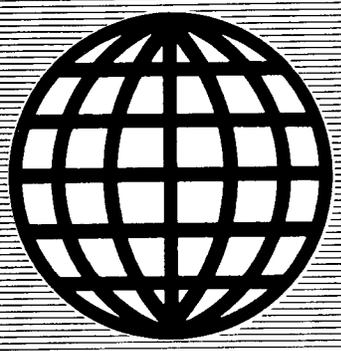


The COMPUTER JOURNAL®

Programming - User Support
Applications

Issue Number 40 September / October 1989 $3.00

Programming the LaserJet
The Escape Codes

Beginning Forth Column
Introduction

Advanced Forth Column
Variant Records and Modules

LINKPRL
Making RSXes Easy

WordTECH's bBXL
An alternative to Expensive Business Software

Advanced CP/M
ZEX 5.0—The Machine and the Language

Programming for Performance
Assembly Language Techniques

Programming Input/Output With C
Keyboard and Screen Functions

The Z-System Corner

Real Computing
The National Semiconductor NS320XX

ISSN # 0748-9331

The Computer Journal

Editor/Publisher
Art Carlson

Art Director
Donna Carlson

Circulation
Donna Carlson

Contributing Editors
Bill Kibler
Bridger Mitchell
Clem Pepper
Richard Rodman
Jay Sage
Dave Wenstein

The Computer Journal is published six times a year by Publishing Consultants, 190 Sullivan Crossroad, Coulmbia Falls, MT 59912
(406) 257-9119

Entire contents copyright © 1989 by Publishing Consultants.

Subscription rates—$16 one year (6 issues), or $28 two years (12 issues) in the U.S., $22 one year in Canada and Mexico, and $24 (surface) for one year in other countries. All funds must be in U.S. dollars on a U.S. bank.

Send subscription, renewals, address changes, or advertising inquires to: The Computer Journal, 190 Sullivan Crossroad, Columbia Falls, MT 59912, phone (406) 257-9119.

The COMPUTER JOURNAL

Issue Number 40 September / October 1989

Editorial 3

Programming the LaserJet 4
Using the PCL escape codes. By Art Carlson

Beginning Forth Column 5
The begininning in a series for those who have hesitated to learn the language.
By David Weinstein.

Advanced Forth Column 10
For those who are ready for advanced topics.
By Dave Weinstein.

LINKPRL 16
Generating the bit maps for PRL files from a REL file. By Harold F. Bower.

WordTECH's dBXL 20
Save money by writing your own custom designed business program.
By Dr. Charles W. Wiley, DVM.

Advanced CP/M 23
Using ZEX, the Z-System executive input processor. By Bridger Mitchell.

Programming for Performance 27
Advanced assembly language techniques for improved performance. By Lee A. Hart

Programming Input/Output with C 30
Using C's keyboard and screen functions.
By Clem Pepper.

The Z-System Corner 36
Remote access systems and BDS C.
By Jay Sage.

Real Computing 41
By Richard Rodman.

Computer Corner 48
By Bill Kibler.

Plu*Perfect Systems == World-Class Software

BackGrounder ii **$75**

Task-switching ZCPR34. Run 2 programs, cut/paste screen data. Use calculator, notepad, screendump, directory in background. CP/M 2.2 only. Upgrade licensed version for $20.

Z-System **$69.95**

Auto-install Z-System (ZCPR v 3.4). Dynamically change memory use. Order **Z3PLUS** for CP/M Plus, or **NZ-COM** for CP/M 2.2.

JetLDR **$20**

Z-System segment loader for ZRL and absolute files. (included with Z3PLUS and NZ-COM)

ZSDOS ..... **$75, for ZRDOS users just $60**

Built-in file DateStamping. Fast hard-disk warmboots. Menu-guided installation. Enhanced time and date utilities. CP/M 2.2 only.

DosDisk ..... **$30 - $45**

Use MS-DOS disks without copying files. Subdirectories too. Kaypro w/TurboRom, Kaypro w/KayPLUS, MD3, MD11, Xerox 820-I w/Plus 2, ON!, C128 w/1571 -- $30. SB180 w/XBIOS -- $35. Kit -- $45. Kit requires assembly language expertise and BIOS source code.

MULTICPY **$45**

Fast format and copy 90+ 5.25" disk formats. Use disks in foreign formats. Includes DosDisk. Requires Kaypro w/TurboRom.

JetFind **$50**

Fastest possible text search, even in LBR, squeezed, crunched files. Also output to file or printer. Regular expressions.

To order: Specify product, operating system, computer, 5 1/4" disk format. Enclose **check**, adding $3 shipping ($5 foreign) + 6.5% tax in CA. Enclose invoice if upgrading BGii or ZRDOS.

Plu*Perfect Systems
410 23rd St.
Santa Monica, CA 90402

BackGrounder ii ©, **DosDisk** ©, **Z3PLUS** ©, **JetLDR** ©, **JetFind** © Copyright 1986-88 by Bridger Mitchell.

Editor's Page

Put the FUN back in Computers

When I first became involved with microcomputers in 1982, I did it because they were fun. I rationalized that I bought the Apple II+ because I needed it for wordprocessing and data management, but the truth is that I wanted to learn what computers could do. We had an active group of enthusiasts who met regularly to talk computers and help each other with problems. It was a very fulfilling hobby. Now, everybody is too busy and too buried in details to spend any time enjoying computers.

We started with small limited machines which provided a lot of challenge and enjoyment. Now we have very powerful large machines which are drudgery. It is time to separate the fun and the work so that we can again enjoy computers as a hobby. It is difficult to dissociate the work-for-profit and hobby aspects when they are performed on the same equipment and in the same setting. That's why many of our readers work on PC-DOS, VAX, or UNIX systems, and then go home and hack on a CP/M or some other system.

Starter PC systems can be obtained for very reasonable prices, but they do not lend themselves to hobby type hacking. They are great for data management if your hobby is stamp collecting, but then again, that is the hobby of stamp collecting and not the hobby of computers. CP/M systems are much more suitable for hobby hacking because an average person can understand and modify the operating system. The very features which make the PC so suitable for business functions are the same features which make it unsuitable for computer hobby hacking. But, we have to face the fact that every year there will be fewer CP/M systems around to serve a greater number of people. We'll have to find a way to enable people with different systems to share in areas of common interest.

Photography is a good analogy. People using 40 year old 4x5 Speed Graphics, 2¼ x 2¼ Bronicas, and the latest 35mm auto everything marvels can discuss common interests such as which films have the finest grain, the fine points of composition, how to be at the right place at the right time for wildlife pictures, etc. They don't spend much time talking about which button to push when, because, after all, the camera is only a tool and they'll probably be using a different one next year. In fact, most of them have a number of different cameras and choose the right one to suit the application.

In computers, we have been cliquish, dividing ourselves by operating system, language, or other minor interest. It is time to change this and to include areas of common interest where people with different systems can participate. This does not mean that we should not include some system specific subjects such as the Z-System crew's excellent series—in fact I would like to do some articles on S-100 or BIOS bashing. But there should also be some areas which are system independent.

System Independence

Because of the current hardware and operating system architectures it is difficult to avoid system specifics. That is a fault of the manufactures because they each try to lock in their customers by using proprietary designs. In the future, there will be much more portability, but we have to work with what we have now (or what we as a group can devise).

I would like to encourage small hardware and software design projects, and one approach to system independence is to avoid the existing systems and use something else. If we assume that most systems have a serial and parallel port we can design projects which interface through the ports. If they don't have the ports we can get system specific for a while and design a way to implement the ports. This way the user can use anything for a platform (Apple II, Mac, PC, CP/M, Atari, Commodore, etc.).

We are preparing several small microcontroller projects which can be wirewrapped. These will involve programming in the microcontroller code and either downloading through a port or blasting a PROM. Providing Hex code, cross assemblers, and/or programmed PROMs will be part of the projects. Learning to use Awk, YACC, and LEX to produce cross compilers which run on various platforms will be part of the overall project for some of us.

Contact me if you want to participate in getting this project off the ground, or if you have ideas for some of the projects.

Small Market Marketing

It is difficult to market books and software or hardware products which are only suitable for a relatively small technical audience. The publishers, distributors, and retailers are only interested in items which will sell in large quantities when displayed in places like B. Dalton's or Waldenbooks. There are excellent products which are very much needed, but which have a more limited special appeal. These products can not justify the expense to advertise in the slick four color magazines, and it is almost impossible to successfully market them under the current conditions.

We are very much aware of this situation, and have decided to publish and distribute some selected products. These products will be plainly packaged, there won't be any fancy advertising, and there will be a minimum of technical support—but we will provide products which would otherwise not be available.

The first products will be Hawthorne's K-OS One generic 68000 operating system plus their cross assemblers. This decision has been made after this issue was laid out, so there is just a small announcement squeezed in.

We are interested in contacting the authors of similar products which can benefit from combined low-overhead promotion. Contact me if you have suggestions on products which should be considered. •

Programming The LaserJet

Part One—The Escape Codes

by Art Carlson

The Hewlett Packard LaserJet printer is very adaptable. Its built-in fixed width fonts can be used for simple letters, or impressive layouts can be generated with custom fonts and graphics. Printing with the built-in fonts can be done without any programming. It is as simple as talking to a dot matrix or daisy wheel printer. I frequently produce draft copies using the PC-DOS PRINT command or a generic runoff program. Incorporating graphics and proportionally spaced fonts in various sizes requires programming, and can get as complicated as you want to make it.

Full featured page preparation and drawing programs with WYSIWYG (What You See Is What You Get) screen display require a lot of programming effort. I recommend that you purchase PageMaker for page preparation or Coreldraw for PC graphics if you need these capabilities.

My programming interests for the LaserJet are in the area of book publishing and outputting directly from a database. These applications will be primarily text, incorporating some graphic elements for logos, graphic ornaments (dingbats), and simple illustrations. The book publishing programs will be command line driven with no graphics display to the screen, although I may incorporate a counting keyboard display. (A typographer's counting keyboard display shows the set width, leading, selected font, remaining line length, total column length, etc.) The database programs will be designed for specific applications such as producing catalogs, directories, and invoices and statements from multiple related data files.

The PCL Programming Language

The LaserJet is controlled with Hewlett Packard's PCL (Printer Control Language). It consists of simple commands delimited with escape characters and can be written by any editor which can embed the escape (1B Hex) character. Programming languages such as BASIC, Pascal, or C are not required, although they will be useful for the more complex schemes.

The PCL commands are sent to the printer as an ASCII string without carriage returns, line feeds, or other wordprocessor formatting commands. Returns or linefeeds should be avoided within the control statements because the printer responds to them and performs unwanted cursor movements. This creates a problem when using line oriented editors which insist on inserting formatting codes. I use WordStar v4.0 in non-document mode for short test programs, but something better is needed for larger working programs. If you have problems with your PCL programs, use debug or a Hex/ASCII dump utility to check for embedded C/R or L/F code within the PCL command statements.

While returns and line feeds should be avoided within the PCL control statements, they can be used within the text to be printed—but there is another way to move the cursor to the next line. I wanted to see how PageMaker handled this, so I used the "Print To Disk" function which sends the output which would normally go to the LaserJet to a disk file instead. This is a very useful feature for studying the PCL language, or for transferring the files to a different system which serves as a print spooler (even an old CP/M system can spool the file to the printer). PageMaker uses absolute X and Y cursor positioning commands instead of returns and line feeds to move to the beginning of the next line. Absolute positioning is much better than returns and line feeds where each line may have a different line spacing and margin.

I would like to suggest a PCL command editor which the readers can use without having to do any programming in BASIC, C, or some other language—but there is the problem with returns and line feeds. The commands are difficult to read without them, and the commands don't perform properly with them. One workaround would be to reposition the cursor after any PCL commands. This would be awkward, but would work except near the bottom of the page where the line feeds may have forced a form feed to a new page. Another alternative is to use the editor's search and replace to remove returns and line feeds from the command sections—or from the entire file if cursor positioning commands are used for new lines.

I will probably write a C program which reads a header file for the commands, and then sends the file to be set, stripping all returns and line feeds. I'll eventually add light bar (or mouse) menu selections for setting the PCL command structure. But I don't intend to provide WYSIWYG display of the file. C Source and executable code will be available, but it will be a while before it will be ready in usable form.

Cursor Positioning

Although the LaserJet does not actually have a cursor, the cursor position refers to the currently active printing position—similar to the cursor on a CRT terminal. The cursor can be moved anywhere within the logical page using a combination of horizontal and vertical positioning commands.

The cursor can be positioned to either an absolute position, or a position relative to its current position. The absolute position is referenced from the upper left hand corner of the logical page area (0X,0Y). Commands are included to to Push/Pop up to 20 cursor positions.

The cursor position can be specified in three different units—dots, decipoints, or rows and columns.

For the LaserJet II, one dot equals 1/300 inch, which is the smallest printable unit.

A decipoint (1/10 of a point) equals 1/720 inch.

The width of a column is specified by the current HMI (Horizontal Motion Index) command in units of 1/120 inch, and the distance between rows is specified by the VIM (Vertical Motion Index) command in units of either 1/48 inch or a selection of lines per inch.

Logical Page

The LaserJet (like most other sheetfed printers) can not print to the very edges of the paper. The physical size, for example

(Continued on page 45)

Beginning Forth Column

Introduction

by David Weinstein

Forth has an odd reputation in the community. It is viewed either as the True Meaning and Salvation and Sacred Writ, or as a peculiar and rather useless language pushed by a group of wild-eyed fanatics. What Forth is is in fact a bit of both. It is a language built around a concept. Unlike other such concept languages (Lisp and Prolog being excellent examples) it is fast, does not require massive amounts of equipment to do something useful, and is fairly easy to implement. The problems the language has had (other than those caused by overzealous programmers) tend to be due to this inheritance (as are the benefits, but we'll get to those in a bit). Because it is easy to implement, the language has proliferated (if it is not the first high level language ported to a new architecture, it is one of the first). However it also means that people's first experience with the language is from a public domain implementation, which in the past meant few of the bells and whistles people had come to expect from computer environments. (Although a Forth fanatic will instantly reply, quite correctly, that many of these are easy to implement, they aren't implemented if the programmer is too disenchanted to ever learn the language). To further frighten a newcomer, the implementations generally only supported the Forth standard for source code manipulation, screen size one K files known as blocks. To add insult to injury the language insisted that programmers give up the infix notation carefully drilled in by succeeding generations of instructors and instead resort to RPN notation, not only for the math, but for everything. Times have changed a bit in the Forth community. For those using computers based on the IBM PC architecture, there is a public domain (not Shareware) version of Forth known as F-PC. Not only is F-PC an excellent professional quality implementation of Forth, it comes with many of the bells and whistles that people have come to expect. And it doesn't use those blasted blocks (in fact, it can use any editor the programmer cares, or the internal SED editor can be used). As for the last well . . .

Why use Reverse Polish Notation anyway?

The "traditional" compiler takes the arguments given and (in general) allocates space on the stack for local variables. One of the features of Forth that makes it so powerful is that the innards of the interpreter are simple in concept, and it is possible for mere mortals to understand what is going on. Arguments are passed to and received from functions (Forth 'Words') by being placed on Forth's parameter stack. Unlike most compilers, which combine data and subroutine threading on a single stack, Forth provides two. The return addresses for subroutine threading are kept on Forth's return stack, while arguments are kept separately on the parameter stack. This means that unlike a conventional language, in which (for the most part) returning multiple arguments or passing or returning variable numbers of arguments is a difficult task, in Forth it is simple. Forth words are in fact generally commented with a "stack picture" out to the side of the definition. This picture is used to describe the state of the stack before and after the word is executed. So (after this quick introduction to Forth mechanics) Forth doesn't use RPN to be difficult, it uses it because it is elegant, simple, and it fits neatly into the Forth paradigm. Similarly Lisp uses prefix notation because conceptually in Lisp a function is merely a special kind of list. However, languages such as Pascal, C, and Prolog have chosen to turn away from the functional model and either make special cases for certain unary, binary, and even trinary operators. Not only do these complicate the compiler, they bring it another level away from the programmer. A Forth programmer can feel free to redefine + (pronounced "plus") which is the addition operator if that is his or her heartfelt desire.

Passing Arguments to and from Forth words

After that longwinded theoretical explanation of why Forth uses RPN notation and how it works, here are some examples to give a feel of why and how it works. Here are the functional definitions of a few Forth basics:

```
+     ( n1 n2 -- n1+n2 )   Add two numbers
-     ( n1 n2 -- n1-n2 )   Subtract one number from another
*     ( n1 n2 -- n1*n2 )   Multiply two numbers
/     ( n1 n2 -- n1/n2 )   Divide one number by another
.     ( n1 -- )            Print the number at the top of
                             the stack
```

So to add two numbers and print the results the Forth phrase would be:

```
5 7 + .
```

To multiply three numbers and subtract the results from one thousand and print the result:

```
1000 20 367 5 * * - .
```

Note that in this case these are all integers. There is a reason. Although Forth is an extremely weakly typed language (it doesn't care what you pass to Forth words, and it doesn't care how you handle storage), it has rigidly typed operators. The * word only multiplies integers (on standard Forths these are 16 bits wide, on some Forths they are 32 bits wide). For the mathematical words in Forth, there are separate words for single integers, for double width integers, and (for those systems that have them) for floating point, as well as for combinations of the above. Again this is a function of the language's simplicity of concept.

How the Forth Interpreter works

For a while I've been discussing the Forth "concept." I think at this point I need to be bit more clear on the issue. As I said, the Forth interpreter concept is understandable by mortals. If Forth is interpreting (which is all we have been doing so far), when it encounters a word on the input stream it first tries to find the word in the dictionary. If it does, then the word is executed. If not, Forth attempts to convert the word into a number (using the current base), and if the attempt succeeds, the number is pushed onto the parameter stack. If it fails, an error message is generated. If we are compiling new Forth words (more on that in a bit), Forth again tries to find the word in the dictionary. Then, if the word is flagged as immediate in the dictionary, it is executed. If the word is not immediate, it is compiled into the dictionary at the first free location (which places it in the current definition). How the compiling is done is dependent on the implementation. If the attempt to find the word fails, then again an attempt is made to convert it into a number. If the conversion attempt succeeds, then the number, and the code to push it onto the stack at runtime are compiled into the dictionary. Otherwise again an error will be reported, and the compilation aborted.

What this means (aside from the fact that the parser is wonderfully simple) is that all Forth words, whether they come with the system or are defined by the programmer, are treated in the same manner. Although this does seem awkward at first, with a bit of practice, Forth's RPN notation becomes second nature. (I know . . . I know . . . it doesn't help when you are learning the language, but it's all I can say . . .) The other side effect of this simple parser is that words MUST be defined before they are used by other words. (Pascal programmers will have no problem with this but those who have become used to two-pass compilers may have a bit of trouble).

Writing your own Forth words

A bit back there were some simple examples of Forth code. But repeatedly typing Forth primitives is not only time consuming and annoying, it is also quite inefficient (in terms of program size, both source and executable). But let us assume that we want to add two numbers, and print the result. When defining a Forth word, you write what is known in Forth parlance as a "colon definition". To define a new word, you start with the Forth word (appropriately enough) : ("colon"). After this the name of the word you wish to define, and then the Forth code which it performs. Finally, the definition is ended with a ; ("semicolon"), much like other high level languages. So our definition would be:

```
: myplus  + dup . ;
```

The word dup makes a copy of the top of the stack and places the duplicate itself on the stack. In this case the duplication is necessary because otherwise the word . ("dot") will print the number AND remove it from the stack. Were we writing this in a Forth program (more on how to write code and save it later), we would define it like so:

```
: myplus        ( n1 n2 -- n1+n2 )
                ( Side effect: Print out n1+n2 )
        + dup . ;
```

As you can probably guess, the word "(" ("left parenthesis") marks a comment. Because this is itself a Forth word (and not a quirk of a compiler), it must be separated from the body of the comment by a space. However, the closing parenthesis is NOT a Forth word. Rather it is a delimiter, and the word "(" merely reads through the input stream until it finds the delimiter, and then exits (again, how this is done will be explained at least in principle a bit later on). This is a case where the ability of Forth to flag a word as immediate is useful, "(" is an immediate word which keeps comments from being handled by Forth (whether or not Forth is in interpret or compile mode).

```
        A Quick Intro To Forth
    (words which are used in the example)

Defining a Forth function (known in Forth parlance as a word):

  : <name> <Forth code> ;

  Fetching the contents of a variable

  <variable> @

  Storing data in a variable

  <data> <variable> !
```

This technique of writing Forth words is one of the most powerful features of the language, and yet it can be misused and be no more useful than the defining of functions in any other high level language. In this case the difference is a matter of style, not of language features. You see, because Forth has dispensed with the myroutine(myarg1, myarg2, . . .) notation which languages inherited from mathematics, it is possible to abstract things out to a degree not possible in other languages. Forth style has as its goal a "noun modifier verb" phrase structure. That is to say, rather than using move (TAPE, drive[2], 50) in Forth it would be proper to say: 50 2 drive tape move. The goal is to hide as much of the "guts" of the language as possible . . . to make the code clear, and then allow the reader to go through layer after layer and see what is going on, but make the purpose clear. Because Forth allows you more flexibility in defining how words work and how things look (this one may end up being explained in more detail in another column), it is possible to define your own syntax to a degree not possible in more conventional languages. Now when I say this is the goal, this is not to say that this is what always happens. For one reason, on most conventional processors subroutine threading is expensive in terms of clock cycles, and so this goal must be sacrificed in the name of performance. For another, this crafting takes work, and takes time. While this makes modification more difficult, it decreases development time. In my experience, it is better to make the initial investment in readable code, because the very flexibility which makes Forth so powerful also makes it quite possible (and even somewhat easy) to write truly horrendous (but working) code.

Saving and loading Forth to and from Disk

This tends to be a problem many people hit head on when learning Forth. And unlike the problems with the language itself, it is more a feature of the implementation than the language, and therefore it is rarely explained in detail in Forth books. It is also at this point that I am going to start differentiating between various Forth implementations. Up until this point, what has been shown should hold true across Forth implementations, but I/O tends to be very implementation dependent. At this point in time, there are two basic schools of thought in Forth. There are those who prefer the original (and only standard) method of Forth I/O, the 1024 byte

Forth "blocks." These 1K blocks are easy to implement, and can be placed on multiple architectures. And there are many in the Forth community who make the claim that, since these 16 line long and 64 character wide screens place a limit on code size, they are useful. And they are correct when they point out that it is good Forth style to keep Forth words short. However, this argument rubs me the wrong way. One of the things I love about Forth is that it doesn't get in my way. It does what I ask it, and it lets me define for myself the terms on which I will deal with the computer. (And if it blows up...well that's my fault). I cannot abide parentalistic languages and compilers which are convinced that they know what I mean. And to me, declaring that because this artificial code segmentation (which, unlike the RPN business is NOT an integral part of the Forth conceptualization) has a beneficial effect when people are first learning the language is grounds for putting up with the flaws of blocks later on is a bit distasteful. The alternative (which has become more and more common in Forth over the years) is for Forth to use the standard files of whatever host the Forth system is sitting on (assuming of course that the Forth is sitting on something and is not running as its own operating system). Unfortunately there are a few partisans of this style of Forth who have gone equally far in the other direction, declaring that no one should use blocks. Now I am not fond of blocks (couldn't you guess?), and I avoid them whenever and wherever possible, but if someone wants to use them, well, that is their business. The "stream" method does have, as one more benefit in its favor, the advantage that it is easy to implement blocks on top of a stream Forth. With that introduction out of the way, many of the older Forth implementations use blocks. These blocks are referenced by number rather than by name, and in many cases the Forth may have example blocks included. A Forth with blocks will use either the word E or EDIT (or both) to edit a block, and the editor will likely be implementation dependent. But to begin editing a block you would type 7 EDIT or 49 EDIT. To show the contents of a block the Forth word is LIST, which takes the same type of argument (that is to say, the number of the block in question). And loading in a block in which you have code is done by using LOAD (again, leaving the block number on the stack before the word LOAD is executed). Unfortunately, I cannot give much help beyond this for block users. The specifics of the editor are doubtless in the documentation (or you had best hope that they are). For people using Forths which allow stream files, you generally have two to three choices. Some, like F-PC, have an editor written in Forth which can be invoked from inside the Forth environment and the editing can be done from there. Others, also like F-PC (it is a very versatile implementation) allow users to run another editor from inside of the Forth environment. And finally, there is the possibility of using another editor entirely which does not displace the Forth environment but which does not interfere with it. This last (which does not really require the help of the Forth) can take many forms: DAs like MockWrite for the Macintosh, TSR editors for the PC, and anything for multitasking systems like the Amiga. Now because stream editors and even more important, the handling of stream files, are not standardized, I am not going to be able to go into detail on the various implementation. For those using F-PC, the internal stream editor is SED, and is invoked by either typed the Forth word SED, or SED followed by the name of the file to edit. Loading a file is done with the word FLOAD followed by the filename (there are other ways of loading files into F-PC, but this is a good place to start). Those using F-PC can also take comfort in the fact that whether in the Forth environment or in SED, hitting <Escape> pops up a top line menu which allows the user to see what is available and to get help.

The code which is written in a block or a file and then loaded is treated (at least conceptually) as if it had been typed from the keyboard. So everything we have done so far will work. Colon definitions will add new definitions to the Forth dictionary, and code outside of definitions will be executed.

More Forth words, and changes in State

There are some words in Forth which cannot be executed in an interpreted mode, but must instead be contained within a definition. In most cases, these are the flow control words such as IF and WHILE (more on these later). At the same time there are some words which are designed for interpreted execution, but not compilation. In previous Forth standards (specifically Forth 79) there were a plethora of "state-smart" words. These would check which state they were in (the STATE variable is available to Forth words) and behave accordingly. The Forth 83 standard moved away from this concept, and began in earnest to split these state-smart words into two words. There is a reason I brought this up at this point. Forth as a standard is very poor in string handling words (although there are many many string packages available in the public domain). One standard, however, is the printing of strings. Those using a Forth 79 system will use the word ." ("dot-quote") at all times, whether in interpreted mode, or inside of a definition. Those using Forth 83 descended systems will use ." inside of definitions only, and a companion .(("dot-paren") in interpreted mode. Now I don't like this system, but because it has been standardized I will explain its use. One of the most common programs written in C is Hello World. To translate this into Forth is simple:

```
Method One (Forth 79)

.'' Hello, World ''

Method Two (Forth 83)

.( Hello, World )

Method Three (Either)

: hi            ( -- )
        .'' Hello, World ''
        ;

hi
```

The first two methods merely execute the code which is given (similar to writing an entire program using only main() in C). The third declares a Forth word (similar to a C or Pascal function or procedure), and then executes the word (similar to executing the function or procedure from the main() routine). While these are simplistic examples, and do not really demonstrate why Forth is a language to learn, they should at least allow you to become familiar with the language and editor.

There a few more I/O words with which you should become familiar before we go on to a few other topics (such as the promised flow control and the declaration and use of constants and variables). The first of these is the word EMIT. EMIT takes as its single argument the ASCII value of a character which it then echoes to the screen. The word bell can be declared using emit with the code:

```
: bell  ( -- )
        7 emit ;
```

This is a bit cumbersome, however. It is impractical to look up ASCII values in a table every time one wishes to print a character. There is a (non-standard but often implemented) word ASCII which is of help here. ASCII takes its single argument differently from most Forth words. You have probably noticed that words like

: take their argument after the word is executed, and this would seem to violate the rule that Forth standard words do not have some special privilege which user defined words do not. In fact it is possible for a word to pull characters out of the input stream themselves. This is how : works, and how any Forth word which needs to get an argument in such a manner (i.e. any word which needs a name which is not defined already). In the same manner, ASCII pulls out a single, space delimited word from the input stream. It then leaves the character on the stack. So to define a word which prints, say, the letter G we would write:

```
: G ( -- )
        ascii G emit ;
```

Forth and Flow Control

So far the code we have written is fairly simple...no branches, no loops. But obviously they are running about somewhere in the language. Before we start working with them, I must warn you that on many Forths, these words can ONLY be used inside of a colon definition. Some of the newer Forths either have made them state smart, or have provided "interpret time" versions of the words. The first case is the IF statement. The traditional form of the Forth IF statement is:

```
IF
        (code)
ELSE
        (code)
THEN
```

As you can probably guess, IF examines the top cell on the stack. If it is true, then the first set of code is executed. If there is an ELSE clause (as in most languages the ELSE clause is optional), then the second set of code is executed if the statement is false, and either way execution resumes after the THEN. The problem I (and many others) have with this is that although it is in fact an RPN version of a conventional IF statement, the choice of the name THEN to end it makes little or no sense. Fortunately this is Forth. If your Forth does not already provide the alternate name ENDIF (or FI if you are a fan of the Bourne shell), ENDIF (or FI if you truly desired) can be added with the following code:

```
: endif         ( alias for IF )
        [compile] then ; immediate
```

Now as it happens this introduces another concept. IF, ELSE, and THEN are all immediate words (they aren't special parser features, they execute during compilation and handle setting up the branches). We want a word which does the same thing as THEN. We can't define it simply as THEN, on most Forths you will get a "Stack Changed" error, as these words keep the marks for branching on the stack. To compile an immediate word into a definition so that it will run at runtime rather than compile time, the word [COMPILE] is used. But since ENDIF is an alias for THEN, it must also be immediate (and hence the definition of ENDIF as immediate). If your Forth has a word ALIAS (and many do), it would be better to use that, as an ALIAS will in most cases be just a duplication of the header information about a word with a different name, whereas defining ENDIF like this also takes up space for the code.

For an example of using these words, let's use a simple function which converts its input to either -1, 0, or 1 depending on whether it is negative, zero, or positive. The code is:

```
: limiter       ( n -- -1|0|1 )
        dup if
                0< if
                        -1
                   else
                        1
                   endif
        endif
        ;
```

The stack notation of LIMITER indicates that it will always return either a -1, 0 or a 1. The first IF takes advantage of the fact that in Forth, the value for FALSE is 0. So if the word is non-zero, we enter the second IF statement. If it is less than zero, -1 is put on the stack, if it is greater than zero 1 is put on the stack. Since the 0< comparator uses up the argument and leaves a flag which is itself used by IF, the stack is empty for those two clauses, and so the only value on it is either -1 or 1. If, however, the value is zero, then the original zero which was duplicated for the comparison is still on the stack, and so the function returns zero. And this is not an utterly useless function either, this sort of limiter is useful in some models of Neural Networks.

The other key area of flow control involves looping. The first type of looping in Forth uses, appropriately enough, the DO and LOOP words (as well as a word +LOOP). The word DO takes as its arguments the limit and starting index of the loop. The word LOOP adds one to the index and proceeds to repeat the loop, unless the limit is equal to or greater than the the index, in which case the loop is exited and the instructions after it executed. If a step other than 1 is required for the loop, the word +LOOP is used instead of LOOP. +LOOP takes an argument on the stack which is the step to apply to the index. So to work backwards from 0 to -10 the code would be -10 0 DO . . . -1 +LOOP. Note that as with the IF clauses, loop constructs on most systems must remain inside of colon definitions. To make life easier for the programmer, the limit and index are not kept on the parameter stack inside of the loop. In most Forths they are moved to the return stack, although I have heard tell of some Forths which have a separate "LOOP Stack" to hold them. To allow access to the limit and index (and more than just the first limit and index in the case of nested loops), Forth provides a special set of words. These are I, which leaves the index of the most recent loop on the stack, J, which leaves the index of the first nested loop, and K, which does the same for the second. Above three levels of nesting you need to write your own code. An example to show how this kind of nesting can be used:

```
: show-nesting ( -- )
      10 0 do
         0 -10 do
            -10 0 do
               .'' Innermost: '' I .  cr
               .'' First nested: '' J . cr
               .'' Second nested: '' K .  cr
            -1 +loop
         loop
      loop
      ;
```

The only new word introduced here is CR, which prints a CR/LF combination. While SHOW-NESTING is a generally useless word, it should provide a feel for how the simple looping constructs work. The other looping constructs provided are BEGIN . . . AGAIN (an infinite loop), BEGIN . . . UNTIL, which loops until the flag which is left on the stack for UNTIL in each loop becomes TRUE, and BEGIN . . . WHILE . . . REPEAT, which executes the section between BEGIN and WHILE at least once, and executes the code for the BEGIN . . . WHILE . . . REPEAT clause as long as the flag on

the stack for WHILE remains true. These words, along with the IF clauses and the DO . . . LOOP construct form the standard Forth flow control section. Some Forths, such as F-PC, also provide a FOR . . . NEXT construct, and most Forths provide some sort of CASE (or SWITCH) statement (and if the one you have does not, there are a plethora of public domain CASE implementations).

Constants, Variables, and dealing with Memory

So far everything we have done has used only the stack for the storing and passing of information. Not only can this not always be done easily (and the byword of Forth is to keep things simple), but in many cases trying to do too much too fast with the stack actually makes the code longer, slower, and much more complex than if a judiciously chosen variable had been used. Defining a variable in Forth is a simple matter. If, for example, we need a variable to hold a pointer to some dynamically allocated memory, we would declare:

```
variable ptr.to.memory
```

Note that there is no special pointer type, we just chose the name appropriately. Variables in Forth are typed only by their size (variables can be one cell wide [usually 16 bits], one character wide, or two cells wide). Since the addresses in Forth are only one cell wide, a VARIABLE can serve quite well as a pointer (just as in many C compilers an integer and a pointer take up the same amount of space and can be cast to each other). Unlike most languages, however, variables in Forth do not return their values on the stack, they instead place their addresses on the stack. To check and alter the value of a variable, we use the words @ ("Fetch") and ! ("Store"). @ takes the address on the stack, and leaves the contents of that address on the stack. ! takes the value to be stored and the address to which it is to be stored on the stack, and leaves nothing behind. So, for example, to initialize our pointer to 0, we would type:

```
0 ptr.to.memory !
```

By the same token, if we want to see the value of the variable we might type:

```
ptr.to.memory @ .

    or

ptr.to.memory @ u.  ( u. is an unsigned version of .
)
```

Perhaps the best way to get used to variables and @ and ! in Forth is to play around with the system variable BASE first. BASE determines the arithmetic base for all mathematical operations. Setting base to binary (2 base !) will accept numbers only in binary, and display numbers only in binary. Similarly Octal, Hex, and any other base you care to try are also available. This easy numerical switching and the inclusion with most Forths of an internal assembler to write Forth words in assembly make Forth one of the best systems I can think of for learning Assembly Language. (In fact I generally learn the assembly language for a new chip by getting a Forth for it and writing small words first).

Constants in Forth are different from variables in two important ways. They are initialized when they are created, and they return their values rather than their addresses. So to define NIL as 0, we would type:

```
0 constant nil
```

Whenever in later code the word NIL is executed, it will leave the value 0 on the stack.

Some Final Thoughts, Hints, and Tips

This article has been meant to give a brief overview of many of the simpler features. By its very nature topics have been covered briefly, and few examples given. The characteristics which make Forth so very very powerful (namely the defining words) have been left for another time. Those who are already hooked should look at the companion article in this issue, which is aimed at those with some experience with Forth. It contains the complete code (and explanations) to add Pascal style RECORDS (complete with nested and VARIANT records) to Forth, and the code and explanation to add Modula-2 style modules to the language. For a deeper introduction to the language I highly recommend "Starting FORTH," by Leo Brodie and published by Prentice-Hall. Be sure to get the second edition, as the first concerns the Forth 79 standard as opposed to the Forth 83. For a style guide, "Thinking FORTH," by the same author is superb. I would very much like to hear from you (comments, questions, recipes, death threats), and I can be reached either by the US Mail at 9036 N. Lamar #274, Austin, TX 78753, or for those with various electronic connections as:

olorin@walt.cc.utexas.edu (Internet)
DHWEINSTEIN on GEnie
Olorin (User #216) on Flight

For those on Usenet I regularly read comp.lang.forth, and for those with GEnie accounts, not only is the Forth RT a wonderful resource, but it has a CATegory (13) which has been set aside for discussion of this Forth column (thanks!). Sending electronic mail is a much better way to keep in touch (the desk doesn't remind me that I have mail buried under 4 inches of debris), and generally gets answered faster than paper mail. •

Forth Resources On-Line

GEnie - (800) 638-9636 (for information) Forth resources in the Forth (page 710) and Mach2 (page 450) Roundtables
Bix - (accessed via Tymnet) Forth resources on Bix can be reached by typing "j forth"
CIS - (800) 848-8990 Forth resources from Creative Solutions, Inc (!GO FORTH), and from Computer Language Magazine (!GO CLM)
Well - Reachable from CompuserveNet or (415) 332-6106 (contains a Forth conference)
Wetware - (415) 753-5265 (contains a Forth conference)
ECFB - (703) 442-8695 East Coast Forth Board
BCFB - (604) 434-5886 British Columbia Forth Board
RCFB - (303) 278-0364 Real-time Control Forth Board
LMI BBS - (213) 306-3530 Laboratory Microsystems, Inc. BBS
DrumaBBS- (512) 323-2402 Druma, Inc. BBS
Flight - (512) 794-8511 (soon to have a Forth conference) This BBS runs on =M=C=D=8, a multi-user BBS from FYI written in a custom version of CONVERS (a Forth descendent)

Advanced Forth Column

Variant Records and Modules

by David Weinstein

This particular column is directed a bit more at intermediate Forth programmers. Since explaining every bit of Forth syntax or providing a stack description for standard words would take more space than the rest of the column, if you aren't familiar with Forth it would help to have an introductory work (such as *Starting Forth*) with you.

Programmers familiar with Pascal or C become used to the ability to define groups of data structures (Pascal Records or C Structures). Although Forth provides the ability to define your own data structures in a much more flexible manner, standard Forth does not provide a higher level interface (such as the Record/Structure) out of the box.

In assembly language, in most cases, equates are used to provide offsets into the data region and are used as selectors (again, computer jargon for methods of referencing data). These selectors are essentially used as an index. The base address of the data is added to the offset, and then the resulting address is the result of the data.

If we were to do this in Forth, the code would look something like this:

```
0 constant NameSize
2 constant NameBuffer
66 constant Income
      ...
create Client allot 78
```

This code does the same thing, however it does not take advantage of Forth's features. To use this code, we would, for example, type:

```
Client Income +
```

This would leave on the stack the address of the Income field of the record (record.Income for you Pascal or C fans). To place the value of this field on the stack, we would use the phrase:

```
Client Income + @
```

This code is still awkward. People learning Forth might argue that using the character @ (pronounced 'fetch') to read in a value from an address is implicitly awkward, however, with a bit of Forth programming practice it becomes second nature. And of course, if you don't like it, the phrase:

```
: fetch @ ;
```

solves the problem. But at this point we are still essentially coding in pseudo-assembly language, lending credit to the assertion that Forth is no more than an obscure dialect of assembly, or a preprocessor. Forth however provides an elegant method for the creation of "templates." These "defining words" allow programmers to create a Forth word which not only creates other Forth words, but defines both their creation time and runtime actions. So, since we have decided that these offsets are in fact indexes into a data block, we will create a defining word called index.

```
: index         ( Creation time: <name> |  offset -- )
                ( Runtime: base -- base+offset )
        create
                ,
        does>
                @ +
        ;
```

The comments between the parentheses are just that, comments. In this case they describe what the word requires in the way of parameters. For my code, words in angle brackets are expected on the input stream. The vertical bar separates the comments of expected input from the comments declaring the stack status of the word. These "stack pictures" are critical to understanding Forth code, because they allow programmers to see the effects each word has on the stack. Words which consume arguments are data sinks (although they may have side effects, in which case they share some characteristics with filters); words which produce data are data sources, and those which alter data or act upon it are filters. In this case, the creation time action of the word index is a data sink with side effects. It will create the word specified as <name>, and take the specified offset and "comma it into the dictionary". This last is a bit of Forth jargon which simply means to take a value off the stack and store it into the first spot of free memory in the dictionary (the data heap which Forth uses for almost everything). The runtime declaration specifies what words created by index will do when they are executed. In this case index creates data filters. They take as an argument the base address and add their offset to it, leaving the result on the stack. So rather than our earlier example of:

```
Client Income + @
```

we would use:

```
Client Income @
```

In this case we would have defined Income with the phrase:

```
68 index Income
```

But our definition of index is not terribly efficient. Since defining words are generally used less often than the words they generate, we probably want to put as much logic as possible in the create

clause of index. We know that adding a zero offset to a base is always going to return the base itself, and we know that the offset value is always defined before the index word is called, we have an invariant condition. So, we could therefore change index to change its actions based upon the value of the offset. If the offset is zero, then we don't want to create a standard index, we want to instead create a word with no stack affects whatsoever . . . a NO-OP. So the revised logic would be:

```
: index         ( Creation time: <name> |  offset -- )
                ( Runtime: base -- base+offset )
        create
                ( offset ) 0<> if
                        ( create )
                                ,
                        does>
                                @ +
                 endif
        ;
```

NOTE: This would be ideal, however Forth syntax does not allow an IF construct to surround the DOES> clause. Furthermore, all words defined with creat leave their address on the stack at run time.

Now this code looks slightly more complex. The offset is compared with zero, and if it is indeed nonzero, we go ahead and create the standard index we defined earlier. The phrases (offset) and (create) are provided as comments to make code clearer. In this case (offset) is used to show what is being compared to zero, and (create) is used to show that this is in essence a create phrase, with the call to create factored out. At this point we have a word which does nothing. It still costs us. It costs us in dictionary space (only critical in very small or very large applications), and it costs us in execution time. We pay the price when we call the word and then return, because on conventional processors jumps to and returns from subroutines are cycle expensive (this problem is solved on processors like the RTX-2000). But there is still a way around this; we can define the created word as being immediate if it has a zero offset. Since the word we create in this case has no effects whatsoever (no stack effects or side effects), making it immediate does not change the action of the code, it just makes it faster. However, in cases where the selector with the null offset is used (to make code easier to read), the selector will not show up in a decompilation of the code (because it does not exist in the compiled code). Here is the revised code which is valid syntax and avoids runtime overhead by making null indexes immediate.

```
: make.index    ( Creation time: <name> | offset -- )
                ( Runtime: base -- base+offset )
        create ( <name> )
                ,
        does>
                @ +
        ;

: make.zero.index ( Creation time: <name> | offset -- )
                  ( Runtime: -- )
        create ( <name> )
               immediate
        does>
                drop
        ;

: index   ( Creation time: <name> |  offset -- )
          ( Runtime: base -- base+offset )
          ( Runtime [alternative]: -- [if offset is zero] )
          ?dup ( offset ) 0= if
                  make.zero.index
           else
                  make.index
           endif
           ;
```

This iterative refinement of index may seem a classic example of spending too much time optimizing trivialities. However, with a bit of experience the optimizations become obvious, and are faster to code than to describe. Unfortunately saying this doesn't mean much to someone first learning the language. In this particular case however, we are going to use the word index quite a bit more. At this point in development we are a step above assembly language in terms of code readability, but not terribly far up. We still have to handcompute and define all of the offsets for each record we define. What we really want is a way to automate these definitions; essentially we do want the record or structure format provided by languages like Pascal and C. So far we have dealt with common Forth words, and with Forth defining words. But if that were not enough, we are going to declare "second-order defining words", that is to say, words which define defining words (don't worry, it can get worse, nth-order defining words have been written, words which can define words which can define words which can... and so in into infinity). The word record which we will write will be a second-order defining word, we want to define record types, not have to go through the whole definition for each instance of the record.

The offset of a record is the size of the record up to the current point. So if the current size is known, we can automatically define the offset. In fact, if we assume that the size of the record up to this point is on the stack, we can very simply define the word element:

```
: element   ( Compile Time: <name> | record.size )
                ( element.size -- new-record.size)
            ( Runtime: base -- base+offset )
        over index
        ( record.size ) ( element.size ) +
        ;
```

Editor's note: The first line in the above code was folded to fit the column width.

In this case, we will assume that the size of the record up to the current point is always kept on the stack (it is initialized to zero by the word record). So if we were defining a record element, the phrase used would be something along the lines of:

```
\ A convenient alias for THEN
: endif
        [compile] then ; immediate

\ "Cell" words to make code more portable
\ (Values set for F-PC)
2 constant cell

: cell+         ( n -- n+cell )
        2+ ;

: cell-         ( n -- n-cell )
        2- ;

: cell*         ( n -- n*cell )
        2* ;

: cell/         ( n -- n/cell )
        2/ ;

: cells+        ( n1 n2 -- n1+[n2*cells] )
        cell* + ;
```

Nonstandard Code

```
64 element NameBuffer
```

Specifying the size is not only easier for the programmer, it also makes the resulting code clearer. The word over is a standard Forth word, which copies the second item down on the stack (in this case the current size of the record. We could make the code more readable to people unfamiliar with Forth (at the cost of some efficiency) by defining a word offset as:

```
: offset
    over ;
```

Or, if the facility is provided in the Forth implementation, we can keep the code efficiency at the cost of some dictionary space, by declaring offset to be an alias of over. (The word alias is provided in some Forth systems as a way to duplicate dictionary entries with minimal or even no runtime cost). Whether or not this choice would be made would depend on the programmer, the application (is size and/or code speed or efficiency a critical consideration, or is slightly more readable code more important), and on whether or not we need the name offset for something more important (possible namespace clashes should be considered in cases like this). In this case, I would leave the code as is, because element is such a short word.

Now we have a mechanism for defining elements of a record, but we do not have the code to define the record itself. And before we lay down the code, we need a clear definition of both the structure of the record, and how we plan to implement it. The elements are already defined as indexes (which makes them Forth words in their own right), and these indexes define the internal structure of instances of the record. Since the elements (or fields) of the record are defined as separate Forth words, the information which they contain does not need to be duplicated in the record template. In fact, all we do need in the way of data for the record template is the size of the record. But this size is not known when the record template is created (if we chose to create the record template when the word record is executed), it is only known at the end of the definition (it is left on the stack by the word element as either the offset of the next element or the final size of the record). So we want record to leave two things on the stack, the address of its sizefield (which we do know), and the constant zero to serve as the starting size of the record. At the same time it should also create a defining word which is capable of using the record size to create an instance of the record. The code to do this is:

```
: record  ( Compile Time: <templatename> | -- size.addr 0 )
          ( Template runtime: <instancename> | -- )
          ( Instance runtime: -- base.of.record.data )
        create ( <templatename> )
               here cell allot 0
        does>
               create ( <instancename> )
                      @ allot
        ;
```

The Forth word here returns the address of the first free byte of memory in the Forth dictionary. In this case it will be the address of the size field. This space is then reserved by the phrase "cell allot", which allocates enough space in the dictionary for the size field. When a template is executed, it creates an instance of the record. When words generated by the Forth word create are executed, they have the address of their body on the stack. In this case the body of a record contains its size. So the address of its size is on the stack when the template is executed, and the template fetches its size, and allocates enough memory for the body of the record itself. You will notice that there is no does> clause for the second create. It isn't needed. If no does> clause is specified, the default action is to leave the address of the body on the stack, which is precisely what we want.

We now have enough information to define the word end-record, which we will use to end a record definition. All this word needs to do is to set the sizefield of the record template, and the information required is already on the stack. So the (rather simple) definition of end-record is:

```
: end-record            ( sizefield size -- )
       swap !
       ;
```

After all of the defining words, and higher order defining words, end-record is refreshingly simple. With the code already defined, we can now define records in Forth. The example could be coded as:

```
record CustomerRecord
    2 element NameSize
   64 element NameBuffer
    2 element Income
    2 element Age
    2 element Height
    2 element Weight
end-record
```

This record definition can be used like so:

```
CustomerRecord WorkingCustomerRecord
```

With individual elements referenced with code as simple as:

```
34 WorkingCustomerRecord Age !
```

Although we have not kept the dot notation used in C and Pascal, we have gained record definitions, and the resulting code uses Forth's "Noun Modifiers Verb" format...a coding style I personally find preferable. But we also might want to nest records. This is no problem at all, because of the implementation method chosen. We don't need to know the structure of an element, just its size, and as each element is calculated as an index, rather than a fixed address, nesting presents no problem at that level either. Since the size of a record is stored in the template body, all we need to use records as elements in other records is a means of extracting the size. Stealing a beat from C, we will define a word sizeof :

```
: sizeof        ( <record name> | -- size )
       ' >body @ ;
```

There is a problem with this code. It isn't necessarily portable. There is no "standard" way to go from the code field of a Forth word (which is what the word ' [pronounced 'tick'] returns) to the body of the word (in which the size is stored). In F83 descended systems, the word >body is used to convert code field addresses to parameter field addresses (otherwise known as the body of the word). But this isn't standard. There is almost certainly a way to do so in whatever Forth you may use, but if it is not documented you made need to go digging into the innards of your Forth. However, the word sizeof does allows us to include Forth records in other Forth records. An example would be:

```
record CaseFile
          6 element Date
sizeof CustomerRecord element Client
       2 element CaseNumber
end-record
```

But there is still one feature we are missing. Both Pascal and C

provide a method of defining records with variable configurations. In Pascal this is called a variant record, in C, a union. In both cases the effect is the same. It is possible to reference the same section of memory, but have it allotted in differing sizes, for differing purposes. In cases where different data may be needed depending on the case (for example marriage data is not needed for single people), we can keep the data in the same record without wasting space.

To implement variant records is actually quite simple with the implementation chosen. The variant portion of a record will have the size of the largest variant clause. In this case, let's take a look at the syntax we'll use first:

```
record Client
      6 element BirthDate
   sizeof String element Name
      1 element MarriageStatus
          variant: ( Based on MarriageStatus )
              ( Married )
              6 element MarriageDate
                  variant: ( Divorced? )
                      ( yes )
                      6 element DivorceDate
                  or:
                      ( Separated )
                      6 element SeparationDate
                  or:
                      3 element SpousePointer
                  end-variant;
              or:
                  ( Single )
                  2 element SOPointer
              end-variant;
end-record
```

From the goal example above, the variant records have to be able to support more than two clauses, and to allow nesting. You also may have noticed that end-variant; has a semicolon at the end, while end-record does not. This is a case of naming choices (I'm not fully satisfied with these names, but I haven't been able to think of anything better). I wanted to be able to use the word or as a separator of clauses. But or is already a Forth word. So I added a colon to variant and to or. But as a general convention in Forth, words which contain a colon and signify the beginning of a segment of code have a semicolon in the corresponding clause ending word. Of course if you don't like the naming choice, changing it is no problem.

To implement the variant records, we need to save the size of the record before the variation (this is the offset to the first element of EACH variant clause, they all start at the same location because they overlap). We also need to keep track of the clause with the largest size, because the entire variant section will take up as much space as the largest clause. From the above specifications, we know that the words variant: and or: have to leave the same type of arguments in the same order on the stack (because or: can follow either variant: or another or:). We also know that the word element requires the "size" of the record on the stack to generate the offset, and therefore the topmost stack element left by both variant: and or: must be a copy of the size at the start of the variant clause (a copy because we need to keep the original for other clauses). This also means that at some depth the words variant: and or: need to

```
\               Record Package for F-PC 2.25
\                written by David Weinstein
\
\ Usage:
\
\ Defining a simple record type:
\
\    record NameString
\         2 element NameSize
\        64 element NameBuffer
\    end-record
\
\ Using records as elements of other records:
\
\    record Client
\                        6 element BirthDate
\       sizeof NameString element Name
\                        7 element PhoneNumber
\    end-record
\
\ Using variant clauses in defining record types:
\
\     record Client
\                        6 element BirthDate
\       sizeof NameString element Name
\                        1 element MarriageStatus
\                             variant: ( Based on MarriageStatus )
\                                ( Is or was married )
\                                6 element MarriageDate
\                                     variant:
\                                        ( Divorced )
\                                        6 element DivorceDate
\                                     or:
\                                        ( Seperated )
\                                        6 element SeperationDate
\                                     or:
\                                        ( Still Married )
\                                        6 element SpouseBirthDate
\                                        sizeof NameString element
Spouse
\                                     end-variant;
\                                or:
\                                   ( Single )
\                                   sizeof NameString element
SigOtherName
\                                end-variant;
\                           7 element PhoneNumber
\     end-record
\
\ Using a record type and record:
\
\     Client CurrentClient
\
\           ...
\
\    : MarriageStatus?  ( -- status )
\       CurrentClient MarriageStatus c@ ;
\
module Records

        : make.index     ( Creation time: <name> | offset -- )
                         ( Runtime: base -- base+offset )
                create ( <name> )
                         ,
                does>
                         @ +
                ;

        : make.zero.index ( Creation time: <name> | offset -- )
                           ( Runtime: -- )
                create ( <name> )
                         immediate
                does>
                         drop
                ;


        : index          ( Creation time: <name> |  offset -- )
                         ( Runtime: base -- base+offset )
                         ( Runtime [alternative]: -- [if offset is
```

```
zero] )
                ?dup ( offset ) 0= if
                        make.zero.index
                 else
                        make.index
                 endif
                 ;

        : record        ( Compile Time: <templatename> | --
size.addr 0 )
                        ( Template runtime: <instancename> | -- )
                        ( Instance runtime: --
base.of.record.data )
                create ( <templatename> )
                        here cell allot 0
                does>
                        create ( <instancename> )
                                @ allot
                ;

        : element   ( Compile Time: <name> | record.size
element.size -- new-record.size)
                    ( Runtime: base -- base+offset )
                over index
                ( record.size ) ( element.size ) +
                ;


        : end-record            ( sizefield size -- )
                swap !
                ;

        : sizeof        ( <record name> | -- size )
                ' >body @ ;

        : variant:      ( rec.size -- rec.size max.size rec.size
)
                dup dup ;

        : or:           ( rec.size max.size prev.size -- rec.size
max.size rec.size )
                max over ;

        : end-variant;   ( rec.size max.size prev.size --
max.size )
                max swap drop ;
```

keep the original size on the stack. And finally, since we need to keep track of the size of the largest clause, we need to keep the size of the largest clause up to this point on the stack. Upon entry into the word or:, the size of the record if the clause were used is on the stack. If we compare this with the previous largest clause and keep the larger, we have the new largest clause size. Since there is a Forth word which will return the maximum of two numbers, we want the largest size up to this point to be the second item deep on the stack, the copy of the size on the first, and the original size the third upon exit from both variant: and or:. When or: is entered, the stack will contain the original size as the third element, the largest clause up to this point as the second, and the size of the previous clause at the top. So we can now define both variant: and or:. Note that variant: leaves as the largest size so far the current size of the record (in case of a null variant clause).

```
: variant:      ( rec.size -- rec.size max.size rec.size )
       dup dup ;
 : or:           ( rec.size max.size prev.size -- )
                  ( rec.size max.size rec.size )
       max over ;
```

All either of these words does is manipulate the stack, since the record structure defining word element uses the current size as its offset and leaves the modified size on the stack. Furthermore since the word variant: only requires the size of the record up to that point on the stack, variant: can be used inside of other variant clauses, satisfying our other design consideration. Please note however that since the word element is separate from the variant feature, the structures inside of variant clauses share namespace with all other element names, even in other variant clauses, so that duplicate names cannot be used. Now all we need to do is write the word end-record;, and the package is complete. Since end-record; follows a variant clause, it will have the same entry conditions as or:. Since it marks the end of the variant clauses, it no longer needs to preserve the original size, in fact, it only needs to leave the size of the record using the largest clause on the stack (as was said earlier, variant clauses will take up as much room as the largest clause). So, we can define the word end-record; as:

```
: end-record;   ( rec.size max.size prev.size )
                ( -- max.size )
        max swap drop ;
```

Editor's note: The first line is folded to fit the column width.

The above package adds full records to Forth. Unfortunately all of the indexes are kept as Forth words, which means it is possible for names to clash. Forth does provide a way around this, in the form of Forth dictionaries, but these, as they exist in the standard, are a bit cumbersome to use, and do not allow easy or neat inclusion of words from various dictionaries. This next set of code very simply uses the existing dictionary structure of Forth to create Modula-2 style modules in very few lines of code (14 to be precise). The only feature of Modula-2 modules not implemented is declaring words or variables "private" by use of a definition module. The main use of this package is to allow words to be grouped by function, and prevent name space clashes.

The first place to start is a defining word MODULE. This word is going to not only create a vocabulary, it is going to make that vocabulary the CURRENT vocabulary (the one in which new definitions are entered). But to do this we need to first save CURRENT and CONTEXT (the dictionary searched for new words), create the vocabulary, execute it (which makes it the CONTEXT vocabulary), and execute DEFINITIONS, which makes it the CURRENT vocabulary. To do this we are going to use one non-portable word, LAST. LAST (and some incarnation of this word exists in every Forth) is in this case a variable which contains the name field address (NFA) of the last word in the dictionary...which will be the word we have just created as the new vocabulary. So by using this address we can execute the most recently defined word. With this problem solved, here is the code to start a module:

```
: module        ( <name> )
        context @ current @
        vocabulary
        last @ name> execute definitions
        ;
```

Ending a module is fairly simple, we just need to restore the pre-module values of CONTEXT and CURRENT (which were stored on the stack to allow nested modules). This is done with the word END-MODULE:

```
: end-module
        current ! context ! ;
```

The slightly more complicated code concerns the use of words from other modules. The syntax we will be using is:

```
FROM <module> IMPORT <name> AS <name>
                     ...
              IMPORT <name> AS <name>
END-IMPORTS
```

We know that FROM must save the values of CURRENT and CONTEXT so as to preserve the environment. And we also know that after FROM and the module name (which will make the module the CONTEXT dictionary), the stack conditions (at least at the top of the stack), must be the same as the conditions when IMPORT is entered after an AS clause. So what IMPORT will do is save the current CONTEXT (which is the CONTEXT of the module from which we are importing). It does this because we are going to need to change the value of CONTEXT when creating the imported word to avoid the (harmless but annoying) "xxx already defined" warning messages. We also want to leave on the stack for AS (which creates the new word) the code field address (cfa) of the old word. So the code for FROM and IMPORT is:

```
: from  ( -- current context )
      current @ context @ ;
: import ( context -- context )
        ( include-context context cfa )
      context @ over ' ;
```

And now we know the entry conditions of AS. AS gets the copy of the old value of the context, the context of the module from which we are importing, and the cfa of the word to import (although not in that order). So the code for AS, which will create a word to execute the imported word is:

```
: as    ( <name> | include-context context cfa -- )
   swap context !          ( Avoid REDEFINED errors)
   create
      swap context !  ( Bring back the  )
                      ( include  context )
      ,               ( And put the cfa in  )
                       ( was the arg )
   does>
      @ execute       ( Execute the imported word )
   ;
```

Since AS leaves the same stack as FROM (we weren't looking at the lower levels of the stack since these are only copied and never altered), IMPORT will work equally well after both FROM and AS. And so all we need to do is write END-IMPORTS which is:

```
: end-imports ( current context -- )
   context ! current ! ;
```

These two packages are examples of how the powerful defining features of Forth, and the access Forth gives to its innards allow Forth programmers to add powerful (and reasonably portable) features to the language, with relatively little effort. ●

```
\               Poor Man's Module Package
\        written by David Weinstein for The Computer Journal
\            copyright David Weinstein 5/30/89
\
\       License:
\               The holder of this copy is hereby given full rights
\               to use the code for both private and commercial
\               purposes, and to pass the code on, so long as full
\               credit is given to the author.
\
\ Module Usage:
\
\       To define a module:
\
\               module <module-name>
\
\                       ...
\                   Module Body (Forth Code)
\                       ...
\                   <Privacy Declarations (see below)>
\               end-module
\
\       To extract a word from a module:
\
\               from <module> import <word> as <destination>
\                                    ...
\                             import <word> as <destination>
\               end-imports
\

\               Module Definition Words
\ Create a vocabulary and make it the current vocabulary
: module     ( <name> | -- context current )
        context @ current @
        vocabulary
        last @ name> execute definitions
        ;

\ End a module definition
: end-module  ( context current -- )
        current ! context ! ;


\               Module Access Words
: from          (  -- current context )
        current @ context @  ;

: import        ( current context -- current context )
                  ( include-context context cfa )
        context @ over ' ;

: as            ( include-context context cfa -- )
        swap context !          ( Swap contexts to avoid )
                                ( REDEFINED errors )
        create
                swap context ! ( Bring back the include context )
                ,               ( And put the cfa in as the argument )
        does>
                @ execute       ( Execute imported word at runtime )
        ;

: end-imports   ( current context -- )
        context ! current ! ;
```

LINKPRL

Making RSXes Easy

by Harold F. Bower

The September 1982 issue of the now-defunct LIFELINES magazine carried an article on a simple program to create a small RAM disk from TPA memory. This appeared to be a neat thing to program at the time, so armed with my trusty old version of MicroSoft's M80 assembler, I attacked it. The one bugaboo in the whole episode was determining what a "PRL"-type file was, and how to generate one. Since none of my tools contained documentation on "PRL" files, the only answer was to count bytes in the program and manually generate the bit map characteristic of a PRL file type. Having survived one episode of this nature, I do NOT recommend it as a normal programming technique.

The beauty of the simple, yet not totally practical, resident module was again applied in early 1983 when I was preparing for a transfer to Germany. Not wanting to suffer keyboard withdrawal while waiting for my household goods to arrive, I had prepared one of my computers for shipment in a suitcase. Unfortunately, only one disk drive would fit. For those who have used a single-floppy system, you know how much wear and tear is placed on the mechanical parts. The solution was to minimize all unnecessary mechanical movements on the drive by using..you guessed it..a resident module! The technique was to copy the disk directory information to a resident module, and access the memory image instead of the real directory for disk reads. The normal load sequence of; seek to the directory, read directory, seek to first extent, read first extent, seek directory, ESC was thereby reduced to seek to the first extent, read, seek second, etc. An updated version of this module will be used to demonstrate the linker program described here.

Having designed the resident module, the problem once again became one of how to generate the bit map. The remembered pain of the RAM disk program convinced me that there was a better way. Since I had become a little smarter in the meantime, I examined the three basic alternatives; first, assemble and link two versions of the program ORGed at different addresses and run a special program to build a map from the differences (a kludge), second, buy an upgraded linker which would generate PRL files (pay money???), and third, write a linker to generate PRL files directly from the MicroSoft REL file. This latter course was selected since it cost the least and had future potential.

The recent articles in these pages by Bridger Mitchell and others on RSX modules, and Bridger's release of the RSX standard have done much to lend a sense of coherency to the topic of resident modules. In return, I would like to donate this little linker to the same purpose. Size constraints on the amount of source code in these pages prevent listing of the complete code, but LINKPRL.LBR has been made available on the Ladera Z-Node (213/670-9465) and Sage MicroSystems East (617/965-7259). The Disk Directory Buffer program used as an example in the second part of this article is also available on the same systems in SPEEDUP.LBR.

LINKPRL is not intended to replace any of the bigger linkers, and will not link multiple modules or other fancy things. Adding these features is left as an exercise for determined readers. LINKPRL will, however, produce either a standard PRL or COM type file from a MicroSoft format REL file produced by M80, Z80ASM, ZAS, or any number of other assemblers. Furthermore, it is only 2k small!

Before delving into the details of how this program works, some background on PRL files is in order. Bit Maps produced by LINKPRL contain one bit for each byte in the combined Code and Data areas. Loaders for PRL files read a byte from the Code/Data area, and a bit from the Map. If the bit is a "1", the desired offset is added to it. For example, the examples in Figure 1 show what assemblers and LINKPRL produce in response to source code instructions.

SOMETHING is assumed to be a routine in the Code area, and DATA is assumed to be a location in the Data area. The High order address byte in both cases is denoted as relocatable by the "1" bit in the Map. The "LD HL,(0006H)" instruction refers to a fixed address, and is not relocatable as shown by "0" in all Map bits. While the original intent of the PRL files appears to have been to relocate files on page, or 256-byte boundaries, relocation to any byte boundary is often performed by using 16-bit addition on the byte corresponding to a "1" Map bit and the preceding byte.

Design

Writing LINKPRL was a learning experience in the MicroSoft REL format. As such, not all link items are implemented, nor are they needed for the generation of simple RSX modules. When link items are encountered in the REL file which cannot be handled, the linker prints a message identifying the item with any other associated information. From this display, you can easily identify the problem statement in the source listing.

I selected the Z80 processor as the target for this little effort since it was, and still is, among the most popular. Since the Z80 is a

Figure 1:

Source statement LINKPRL input	Assembler out/	Map bits from LINKPRL
LD HL,(0006H)	2A 00 06	000
CALL SOMETHING	CD 00 00	001
CP 32	FE 20	00
LD DE,DATA	11 00 00	001

register-oriented device, and I have a fetish for fast programs (see ZSDOS), LINKPRL makes maximum use of the internal CPU registers. The specific register selections used here have only two known side-effects; no bounds checking is performed on the output code (for speed reasons), and the BDOS/BIOS system must preserve or not use the alternate nor index registers. This constraint is in effect for other programs as well (see "ZSDOS, anatomy of an OS" in issues #37 and #38) and is only known to be a problem on a few systems. For most programs, the lack of bounds checking on output code should also be no problem.

In bit-oriented files, such as the MicroSoft REL format used by LINKPRL, individual bits, rather than bytes, have distinct meaning. This is in contrast with the byte-oriented structure of the Z80 microprocessor and character-oriented (seven or eight bit) text and HEX files. The Z80's expanded instruction set with bit rotates, tests and sets proved invaluable in efficiently handling the bit-oriented files.

As previously mentioned, maximum use is made of the Z80 registers to obtain the smallest code and fastest possible execution times. Register usage within the main body of LINKPRL is shown in Figure 2.

Much of LINKPRL consists of standard routines to Read a file, write a file, receive your input from the console, and print messages to you. These sections will not be covered in detail since numerous examples are available in other documents and programs. The unique parts of LINKPRL are those dealing with processing of the bit-oriented REL files and we will concentrate on these parts.

How It Works

As with most well-behaved programs, LINKPRL begins with code to set a local stack, check for a Help request instead of execution, and check for the presence of a valid filename argument. If no file type is entered, REL is assumed. The specified file is then opened. The next action taken is to clear all available memory from the program end to the base of the BDOS to zeros. This initializes all data areas to a known value (0) as an aid in debugging, and to assist your code reduction efforts by possibly reducing the need to initialize data areas within your relocatable module.

After clearing memory, you are prompted for the mode (COM or PRL), and the starting address which defaults to 100H for each mode if only a Carriage Return is entered. With that, the preliminaries are complete and LINKRL gets down to work at label DEFAULT (Figure 3) where the first file read occurs.

LINKPRL reads a byte at a time from the input file to the C register and shifts the byte, bit-by-bit to recover the command and data elements from the file. GETBIT is the basic routine to get a bit from the file. The Most Significant bit (Bit 7) of register C always reflects the current bit of concern. When a new byte from the source file is required, the byte is loaded into C and an eight count is loaded into the B register to serve as a control to indicate when a fresh byte is required. Each call to GETBIT decrements the counter,

Figure 2:

```
A = General Purpose
B = Bit Count for Source Byte
C = Source byte shifted (B7 is current)
D = General Purpose Counter
E = Bit Map Output Byte
HL = Pointer to Input File
AF' = General Purpose and Byte Transfer
BC' = Program "ORG" Location
DE' = Data "ORG" Location
HL' = 16-bit accumulator for Displacement Calculations
IX = Physical Load Location
IY = Points to FLAGS Byte
```

Figure 3:

```
DEFAULT: POP   HL          ; Get address from stack
   LD (ORGADR),HL
   LD HL,BUFF
   CALL  READ     ; Set bit position
   JR LOOP0    ; ..and jump to test bit

;=============================================;
;     M a i n    P r o g r a m    L o o p    ;
;=============================================;

LOOP: CALL  GETBIT                  ; Get a source bit into position
LOOP0:   JR NZ,LOOP1       ; ..jump if 1x form and test next
   CALL  BYTE0    ; 0 = load 8 bits absolute
   JR LOOP     ; ..and back for more

; We have 1x form.  Check the second bit

LOOP1:   CALL  GETBIT               ; Get a source bit into position
   JR Z,LOOP2     ; ..jump if it is 10x form
   CALL  GETBIT            ; We have 11x, Check 3rd bit
   JR NZ,COMREL   ; 1 = Common, 0 = Data

; We have 110 = Data Relative.

   CALL  ADDR16            ; Get 16 bits, data relative
   EXX         ; Do the math in alternate regs
   LD HL,(TEMP)   ; Load the offset
   ADD   HL,DE    ; ..and add DSEG base from DE'
   JR OUTV     ; Write a 01 to Bit Map

; We have 111 = Common Relative.

COMREL:  CALL  ADDR16               ; Get 16 bits, common rel
   PUSH  HL    ; Write 01 to map
   PUSH  DE
   LD DE,(COMMAD) ; Add Common Base address
   LD HL,(TEMP)   ; ..to accumulated offset
   ADD   HL,DE
   POP   DE
   EX (SP),HL
   EXX
   POP   HL
   JR OUTV     ; Save value & write 01 to Bit Map

; We have 10x form.  Check the third bit

LOOP2:   CALL  GETBIT               ; Get a bit in position
   JR Z,SPECL     ; ..jump if 100 (Special Link Item)

; We have 101 = Program Relative.

   CALL  ADDR16            ; Get 16 bits, prog relative
   EXX         ; ..writing 01 in bit map
   LD HL,(TEMP)
   ADD   HL,BC
OUTV: LD A,L      ; Vector here to output
   EXX         ; ..relative addresses
   CALL  BYTE0V            ; Low byte has 0 Map Bit
   EXX
   LD A,H      ; Get Hi byte
   EXX
```

```
   CALL  BYTE1V            ; ..write with 1 Map Bit
   JR LOOP

; Arrive here if special link (100xxxxxxx)
; We don't do much with these, most just print information

SPECL:   CALL  GETTYP               ; Get 4 bit type
   EXX          ; Swap to alternates to get free HL'
   LD HL,SPLTBL   ; Offset from table start
   ADD   A,A      ; Double value for 2-byte entries
   ADD   A,L
   LD L,A
   JR NC,SPECL0   ; Bypass next if no Overflow
   INC   H
SPECL0:  LD A,(HL)                  ; Addr.low to A..
   INC   HL
   LD H,(HL)      ; Addr.high to H
   LD L,A      ; Complete address to HL
   PUSH  HL    ; Address to stack to simulate CALL
   EXX         ; ..back to primary registers
   RET         ; Jump to Address on stack

SPLTBL:  DEFW  ENTRY                ; 0 = Entry Symbol
   DEFW  COMNAM            ; 1 = Common Block Name
   DEFW  PGNAME            ; 2 = Program Name
   DEFW  SEARCH            ; 3 = Library Search
   DEFW  UNDEF0            ; 4 = (undefined)
   DEFW  COMMON            ; 5 = Common Size
   DEFW  CHNEXT            ; 6 = Chain External
   DEFW  ENTRPOINT         ; 7 = Entry Point
   DEFW  UNDEF1            ; 8 = (undefined)
   DEFW  EXTOFF            ; 9 = External + offset
   DEFW  DATSIZ            ; 10 = Data Size
   DEFW  LODLOC            ; 11 = Load Location
   DEFW  CHNADDR           ; 12 = Chain Address
   DEFW  PRGSIZ            ; 13 = Program Size
   DEFW  FINI     ; 14 = End of Program
   DEFW  FINI     ; 15 = Module End
```

and rotates the byte in C one position to the left.

When the file and the BC registers are loaded, the main portion of the program is entered at label LOOP (Figure 3). This entry point expects a basic identification structure of one or three bits. If the first bit is a "0", the next eight bits form a byte to be loaded immediately into the file. If the first bit is a "1", then the next two bits must be loaded to determine the specific action needed. The three bit field at this point is interpreted as shown in Figure 4.

Special Link items are of particular concern to LINKPRL, even though only a few of the 16 possible items are used. These items specify such things as Names for various program parts, Load addresses (due to assembler ORG directives), and Sizes of the Data and Program areas. These last two items must be recognized before any bytes are sent to the destination file. When both Data and Code size statements have been received, the remaining registers and values are set and program loading begins in earnest.

LINKPRL builds a memory image of the output file, writing it to disk when the entire REL file is read, or either the End of Program or Module End Special Link items are encountered. The image begins with a 256-byte header record (only two bytes used) if a PRL file is being generated. The header record is absent for a COM file. The next section (the first for a COM file) is Program code, followed by Data specified by the DSEG assembler pseudo-op. For PRL files, the byte after the end of the Data area marks the beginning of the Relocation Bit Map and this address relative to the start of the Code area is placed in the first two bytes of the header record. In this manner, loaders can determine the size of the file Code and Data area, and the address of the Relocation Bit Map. For COM files, the end of the Data area marks the end of the file.

```
                Figure 4:

   100   Special Link item
   101   Load 16-bit word, Program Relative
   110   Load 16-bit word, Data Relative
   111   Load 16-bit word, Common Relative
```

As bits are read from the input file, various routines are accessed to write output bytes as required. The principal ones for the purpose of generating PRL files are BYTE0V and BYTE1V (Figure 5) which save a byte value with a "0" and "1" Map bit respectively. For word addresses relative to Code, Data or Common Segments, the single entry point OUTV (Figure 3) is used to send the low-order byte with a "0" Map bit followed by the high-order byte with a "1" Map bit. CHKWRT (Figure 5) is the routine which sets Map bits indicated by BYTE0V and BYTE1V and writes a byte to the Bit Map area when eight bits have been accumulated.

One of the more difficult things to handle in a simple linker such as LINKMAP is the Set Load Location directive resulting from assembler ORG and DEFS (reserve space) directives. The easy path was taken in which a pseudo program counter is maintained in memory, and used to calculate the displacement values resulting from Load Location directives. Null bytes with "0" Map bits are then written until the desired location is achieved. As you can deduce, there is one "gotcha" -- an ORG resulting in a negative displacement will probably fill the entire memory space, including the operating system! Consequently, do not attempt to use LINKPRL on routines which use the ORG directive in a negative direction.

The remainder of the code consists mostly of general routines performing utility functions, and interfacing to the operating system. If you are particularly interested in the internal functioning, download the source and examine away! For the rest of you, let's cover the operational details.

LINKPRL operation is simple (how else could it be kept to 2K?) requiring that a REL file name be entered as an argument to the LINKPRL invocation as:

```
LINKPRL filename
```

An opening banner will be displayed, and LINKPRL will check for a filename entry in the default File Control Block. If a double-slash (//) is detected instead of a filename, LINKPRL displays a brief Help message summarizing the syntax and operation returns to the Command Processor. Any other entry is assumed to be a filename, and LINKPRL will try to open the file in the current User area. Any error in opening the file will return you to the Command Processor.

Assuming that a REL file is successfully opened, you will then be prompted for the remaining two items needed to start the linkage with:

```
Produce .COM or .PRL file (C/P) :
Enter Hex load addr (Default = 100H) :
```

If you enter a "C" or "c" at the first prompt, the specified file will be linked to a file of the same name with an extension of COM. No Bit Map will be included in the output, and it may be directly executed as any other COM file. Entry of a "P" or "p" will create a file of the same name with an extension of PRL. This file will contain the header record and Bit Map described earlier.

For most applications, the default load address requested in the second prompt will be selected with a simple Carriage Return. Special uses where LINKPRL may be used with different values include generating a ROM image, or linking for execution in high

Figure 5:

```
;.....
; Output a byte with a 1 Map Bit

BYTE1V:  SCF              ; Set Carry flag for 1 in Bit Map
   JR CHKWRT      ; ..and do it

;.....
; Accumulate 8 bits into a byte and output with a 0 Map Bit

BYTE0:   CALL  GETBYT              ; Gather 8 bits into a byte
BYTE0V:  OR A     ; Reset Carry for 0 in Bit Map
         ;..fall thru to..
;.....
; Check for output write status on Map Bit
;  Carry Flag unaffected until shifted into E Register

CHKWRT:  BIT   2,(IY+0)    ; Check ok-to-load
   JR Z,CHKWR2 ; ..Error if flag = 0

   LD (IX+0),A ; Save Code byte
   INC   IX    ; ..and bump code pointer

   PUSH  HL    ; Preserve regs
   LD HL,(PCNTR)
   INC   HL    ; Increment Pseudo-Program Counter
   LD (PCNTR),HL

   RL E     ; Rotate Map from Carry into E
   LD HL,COUNT
   INC   (HL)     ; Bump count..
   BIT   3,(HL)           ; ..check = 8?
   JR Z,CHKWR1 ; Exit if < 8
   LD (HL),0      ; ..else reset counter to 0
   LD HL,(BITMAP) ; Write 8 Map bits out
   LD (HL),E
   INC   HL    ; ..bumping address
   LD (BITMAP),HL
   LD E,0      ; Preset next map byte to 0
CHKWR1:  POP   HL          ; Restore regs
   RET

CHKWR2:  CALL  ERRV                ; Print message & Abort
   DEFB  CR,LF,BELL,'Write attempt before areas sized !$'
```

memory such as for ZCPR3 Type 3 modules. For these cases, select the "C" option, and the appropriate starting location.

When you have entered a terminating Carriage Return after the load address prompt, LINKPRL takes off to "do its thing" and will display some informative status such as Module Name, Program Size and Data Size. If linking to a PRL file, the relative starting and ending addresses of the Bit Map will also be displayed. Other than these items, any unrecognized link items will also be displayed. Since this article is already becoming quite lengthy, I refer you to the source code for descriptions of such items.

When the link is completed, you will be returned to the Command Processor prompt. In the next part of this article, we will demonstrate how LINKPRL is used in both COM and PRL modes to generate a Disk Directory Buffer RSX module. In the meantime, enjoy your new mini-linker. •

XED4/5/8 Integrated Editor Cross-Assembler

XED4/5/8 is a fast and convenient method to develop and debug small to medium size programs. For use on Z80 machines running Z-system or CP/M. Companion **XDIS4/5/8** disassembler also available.

Targets: 8021, 8022, 8041, 8042, 8044, 8048, 8051, 8052. 8080, Z80, HD64180, and NS455 TMP.

Documentation: 100 page manual.

Features include:

★ Memory resident text (to about 40 KB) for very fast execution. Recognises Z-system's DIR: DU:. Program re-entry with text intact after exit.

★ Built in mnemonic symbols for all 8044,51,52 SFR and bit registers, NS455 TMP video registers and HD64180 I/O ports.

★ Output to disk in straight binary format. Provision to convert into Intel Hex file. Listing to video or printer. A sorted symbol table with value, location, all references to each symbol.

★ Supports most algebraic, logic, unary, and relational operators. Eight levels of conditional assembly. Labels to 31 significant characters.

★ A versatile built in line editor makes editing of individual lines, inserting, deleting text a breeze. Fast search for labels or strings. 20 function keys are user configurable.

★ Text files are loaded, appended, or written to disk in whole or part, any time, any file name. Switchable format to suit most other editors.

★ The assembler may be invoked during editing. Error correction on the fly during assembly, with detailed error and warning messages displayed.

For further information, contact:

PALMTECH
(a division of Palm Mechanical)
cnr. Moonah & Wills Sts.
BOULIA, QLD. 4829
Phone: 6177 463-109 Fax: 6177 463-198 AUSTRALIA

For Sale

Turbo Pascal—Version 4.0 (**NOT** current version). Never used, complete with manual, original disks, registration card. For IBM-PC with 5¼ disks **$45**

Sheet feeder for Quem Sprint 15 —Single bin feeder in excellent condition. **$50**

Bi-directional Forms Tractors for Quem Sprint 15—Excellent condition. **$50**

Zenith Z-19 Terminal—Needs work (video board ?) **$75**

GX-10 Printer Motherboard—Apparently complete and working **$25**

All prices plus shipping

The Computer Journal
190 Sullivan Crossroad
Columbia Falls, MT 59912
Phone (406) 257-9119

WordTech's dBXL

An Alternative to Expensive Business Software

by Dr. Charles W. Wiley, DVM

Dr. Charles Wiley owns and operates a veterinary hospital in Alice, Texas. He became interested in computers in 1984 and purchased an IBM PCjr. Since that time, he has taken a correspondence course in computer science and has written several office programs in BASIC. He now uses dBXL on an IBM compatible PC-XT with 640K RAM, 20MB hard drive, 5.25 and 3.5 inch floppy drives. He wrote his very functional office program without any outside help. He used as references, "Understanding & Using dBASE III Plus" by Rob Krumm and the Manual for dBXL.

Summary

Trying to find software for a particular type of business can be very difficult and frustrating in the least. Purchasing pre-packaged software is expensive and may leave the buyer with a product that is hard to learn or contain parts that are useless for their applications. WordTech's dBXL will allow even the novice computer operator, the ability to fully computerize their business. The program has very powerful search and retrieve functions and can be programmed, if the user wishes, to include some very fancy screen displays. Plain or fancy, dBXL can do it for you.

WordTech's dBXL

How much would you expect to spend on a custom designed business program? $1000? $2000? More? What are the chances that it will do everything you want it to? Will their program fit your needs, or more likely, will you have to fit your business into their program?

What if I show you how you can custom design your own program, allowing you to input only the information you need and print out only the information you want? How much? How about less than $250. No gimmick, please read on.

There are software programs on the market today that will allow an individual to custom design his/her own business program without investing a small fortune. The one that I am going to talk about is WordTech's dBXL.

Most of us have seen articles for dBASE II, III, and III+. They are the most powerful database programs on the market, right? Wrong. Everything dBASE does, dBXL does better and with less wasted space. dBXL was advertised as a dBASE clone. After using it, I feel that dBASE is a clone of dBXL! To demonstrate how similar dBASE is to dBXL, the textbook that I used to write my program was "Understanding & Using dBASE III Plus" written by Rob Krumm and published by Simon & Schuster, Inc. Now, how about cost? dBXL lists for $249 (you can get it cheaper from a discount software distributor) compared to $499 for dBASE III+.

One important feature that dBXL has is the ability for the user to set up their database(s) and begin inputting information right away. Fancy screens, windows, bells and impressive displays are all possible, if you want them. They are not necessary for the program to be functional. They can be added as you go, later, or not at all. You have the choice of what you want the program to do, not vice-versa.

If you don't think you have enough time or knowledge to do this, think again. I am a veterinarian and I do not have a degree in computer science. I have moderate knowledge in the basics of IBM compatible computers, but by no means am I an expert.

I purchased dBXL and started programming between patients. As I developed more intricate parts to the program, I became engrossed in what it could do and wanted to do more. Even now, although the program works fine, I'll think of something new for the program to do. I'll get out the programming guide and learn to do it. The more you add, the more pride you take in it.

There are going to be some of you out there that don't want to spend any time doing such trivial things. That's okay, you guys are necessary to keep the programmers in business. In fact, call me and I will be glad to write a program on dBXL for you for half of what you will pay for custom programmed software.

Now, let's get a little more specific. I am not going to attempt to give a complete course on programming dBXL or dBASE III+. I do want to give you some pointers on getting started though.

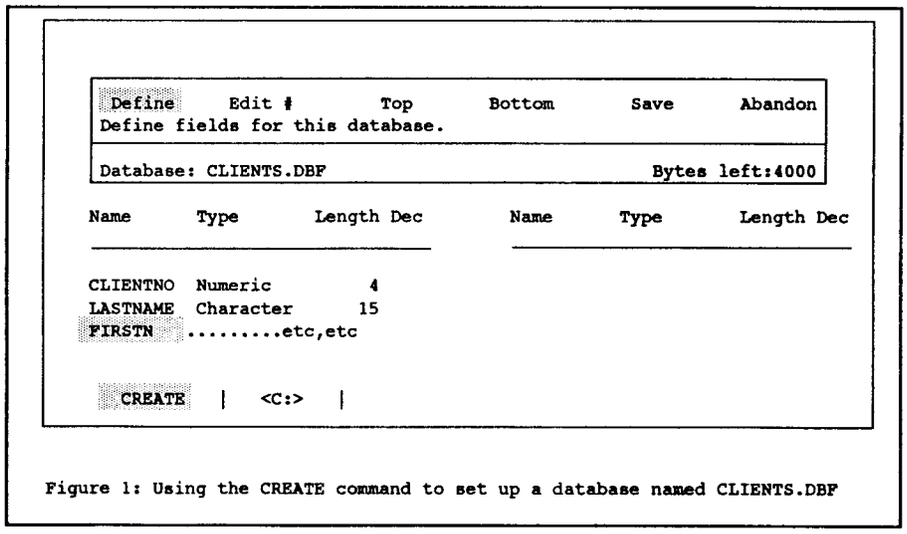


Figure 1: Using the CREATE command to set up a database named CLIENTS.DBF

```
Structure for database : C:\DBXL\CLIENTS.DBF
Number of records      : 1483
Last update            : 05/31/89

Field  Field Name  Type       Length    Dec

    1  CLIENTNO    Numeric         4
    2  LASTNAME    Character      15
    3  FIRSTNAME   Character      15
    4  ADDRESS     Character      20
    5  CITY        Character      15
    6  STATE       Character       2
    7  ZIP         Character      10
    8  PHONE1      Character       8
    9  PHONE2      Character       8
   10  STATUS      Character       1
   11  LASTVISIT   Date            8

** Total **                     107
```

Figure 2: CLIENTS.DBF database structure

```
Structure for database : C:\DBXL\HOSPITAL\PETS.DBF
Number of records      : 728
Last update            : 05/31/89

Field  Field Name  Type       Length    Dec

    1  CLIENTNO    Numeric         4
    2  PETSNAME    Character      10
    3  PETNO       Numeric         4
    4  COMMENTS    Character      15
    5  SPECIES     Character       8
    6  BREED       Character      15
    7  COLOR       Character      10
    8  SEX         Character       2
    9  MATE        Logical         1
   10  WEIGHT      Character       3
   11  BIRTHDATE   Date            8
   12  RABIESTAG   Character       7
   13  VACCDATE    Date            8
   14  OTHERVACCS  Character      10
   15  NEXTVISIT   Date            8
   16  NEXTPROC    Character      20
   17  CLINICAL    Memo           10

** Total **                     144
```

Figure 3: PETS.DBF database structure

```
Record No.      729
CLIENTNO
LASTNAME
PETSNAME
PETNO
COMMENTS
SPECIES
BREED
COLOR
SEX
MATE            ?
WEIGHT
BIRTHDATE         /  /
RABIESTAG
VACCDATE          /  /
OTHERVACCS
NEXT VISIT        /  /
NEXTPROC
CLINICAL          MEMO
```

Figure 4: To enter the memo field, press Ctrl-Pg-Dn

First of all, you must decide what information you want in your database. A well thought out database is imperative to good operation. Each database can have as many as 128 different fields in version 1.2c (this is compatible with dBASE III Plus). Version 1.3 has added the command SET COMPATIBLE which determines whether the maximum number of fields in a database is 128 or 512. Those fields can be either character, numeric, logical, date or memo.

Each field must have a field name that is not longer than 10 characters and must start with a letter. Use field names that are simple and as short as possible. You can always change them later if you don't like them. I will describe another way to have custom inputs and outputs a little later.

Now, after we have decided what we want in our database, we need to set one up. This is where dBXL really makes it easy for novices. They have a built in INTRO program that makes it very easy for new and inexperienced persons to immediately begin using dBXL. Once the program is loaded, a prompt will appear at the bottom of the main screen:

```
XL [1] >
```

At this prompt, type the letters, "intro" and the program will lead you through with very easy menus. dBXL's INTRO program beats the one in dBASE, hands down.

For some of you, this is all you will need to begin and run a program. For others, like myself, there is the desire to create our own visual screens, with our own personal messages, inputs and data outputs. For us, we will want to learn the "command language" of dBXL.

I will briefly describe how I set up my program, giving you some insight into some of the features of dBXL. To set up a database, enter the command "CREATE" at the prompt. Figure 1 shows the CREATE screen where the fields are set up.

I set up three databases, CLIENTS, PETS and TRANSACT. I used separate databases for clients and pets so that I wasn't repeating client information in multiple pet situations.

From Figure 2, you can see that I have only the information that is necessary about my clients. The "lastvisit" field was added so that I can keep track of the last time a client was in the office.

The PETS database (Figure 3) has only the information that I need to keep on each pet. You will also notice the last field called MEMO.

I want to explain the MEMO field in more detail. When we are editing a record in the PETS database, we can place the cursor on the MEMO field and press Cntrl-Pg-Dn (Figure 4).

The user is then put in the dBXL's text editor (Figure. 5). The text editor is a word processor and can be used to store any text that is too long to fit into a field. The data can then be stored under that particular record for retrieval and editing at any time. For businesses that need to input text, this part of the program is priceless. It allows input of text, in any format, up to a total of 10,000 characters. Without the MEMO field, we would have to set up a separate field called "clinical history." Then, a designated amount of space would be reserved in each pet's record for this information, whether you needed it or not. This would be a waste for some pets and not enough for others. Therefore, by using the MEMO field, we can write what is needed and only use the amount of space necessary. It can be used to store any type of information the user wants, within the memory limits. To leave the text editor, we press Cntrl-Pg-Up.

Our last database is entitled TRANSACT. It contains daily transactions. Each file contains the date, client's number, pet's number, and the services they obtained. After entering the services that the client received, the data is printed out on paper and serves

```
dBXL Word Processor   total lines=2    Caps     line=1  col=0
*  *  * top of file *  *  *
5/31/89  Rabies, DA2PL-CPV, Fecal Exam (neg)
         Physical exam= OK

*  *  * end of file *  *  *
```

Figure 5: The word processor allows entry of text. Press Ctrl-Pg-Up to exit and go back to the screen in Fig. 4.

```
PETFORM.FMT           total lines=48           line=1  col=0
*  *  * top of file *  *  *

      @ 0,11 SAY "Alamosa Animal Hospital, Inc."
      @ 1,20 SAY "Pet Information Database"
      @ 3,15 SAY "Client Number:"
      @ 3,30 GET clientno
      @ 3,45 SAY "Pet's Number:"
      @ 3,57 SAY petno
      @ 4,15 SAY "Owner's Lastname:"
      @ 4,30 GET lastname
      @ 5,15 SAY "Pet's Name:"
      @ 5,30 GET petsname
      @ 6,15 SAY "Comments:"
      @ 6,30 GET comments
      @ 7,15 SAY "Species:"
      @ 7,30 GET species
      @ 8,15.....etc,etc.
```

Figure 6: Format file for custom designed screen inputs

```
XL [1] PETS> LIST PETSNAME,LASTNAME FOR SPECIES="CANINE"
.AND. BREED="LABRADOR"

Record#  PETSNAME    LASTNAME

    513  BUDDY       OWENS
    549  CHRIS       VILLARREAL
    575  JAKE        FARRIS
    614  LADY        THOMAS
    629  SUE         DAUGHTRY
    634  SUSIE       NESLONEY
    651  SHAMIE      MATHIS
    658  TAZ         RICHTER
    713  PRINCESS    HOELSCHER

XL [1] PETS>
```

Figure 7: Listing information is dBXL's strongpoint

```
ALAMOSA ANIMAL HOSPITAL, Inc.

       Select one of the following operations:

         A. Add New Clients
         B. View/Edit Client Records
         C. Add Pet Records
         D. View/Edit Pet Records
         E. Enter Transactions
         F. View Transactions
         G. Print Transactions
         H. Print Vaccination Certificates
         I. Print Records
         J. Print Reminders
         Q. Quit and Return to dBXL prompt

              Enter Selection
```

Figure 8: Main menu for Alamosa Program

as a receipt.

Once our databases have been set up with the information we want, we can add to or remove fields at any time. What if we decide we want to add or remove a field? No problem. We use the command MODIFY STRUCTURE, and then add or remove fields. We can also change the name of a field.

If you use the INTRO program to run your program, the data will be entered just as you set uo the field names, vertically down the screen. As I mentioned earlier, if you want to have your own personally designed screens, you can do so by setting up "format screens". I can't go into depth on setting up screens, but I will give a brief description. We use a series of @...SAY...GET commands to position the input areas of the fields anywhere on the screen. I have shown part of my format screen for listing my PETS database on the screen in Figure 6.

As the format starts, it says, @ 0,10 SAY "ALAMOSA ANIMAL HOSPITAL, Inc." This prompts the program to put the cursor at line 0 and row 10 and prints the message "ALAMOSA ANIMAL HOSPITAL, Inc.". The fourth line says, @ 3,30 GET clientno. This positions the cursor at line 3 and row 30 and allows the user to input data for the field of "clientno". Once the data has been inputted with a GET statement, it is stored in the database.

The same thing can be done for displaying the data on the screen or on the printer. For printing purposes, the user must use the command SET DEVICE TO PRINT. As you can see, the possibilities are endless.

The most powerful parts of this program are the many ways you can retrieve the data. Searching for information is quick and easy. For instance, if I were wanting to find all the dogs in my PETS database that were male labradors, I would use the PETS database and enter the following filter command:

```
LIST PETSNAME, LASTNAME FOR SPECIES=''CANINE'' .AND.
BREED= ''LABRADOR'' .AND. SEX=''MA''
```

This would start dBXL searching each of the fields (species, breed, and sex) for that particular information. If it found the information, it would list the pet's name and owner's lastname on the screen for each matching file (Figure 7).

If I wanted the information printed, I would enter the command LIST TO PRINT PETSNAME, LASTNAME, FOR.....etc. The search and find possibilities are outstanding.

The best of the show is yet to come...to get really elaborate, you can use the WSET WINDOW and WUSE WINDOW functions to create windows to display or input data. Window sizes can be varied as well as the information they display. The windows can be overlapped and as many as 99 different windows can be opened on the screen at one time! This makes a very impressive program. Figure 8 shows my main menu. If we enter selection "B" from the menu, then a window screen opens right over the previous screen (Figure 9). If we then enter selection "2" from the second screen, we have a third window screen that prompts us for the name that we want to search for (Figure 10).

There are many more functions to dBXL. My intent was to give a brief overview of what the program is capable of. I hope I have convinced some of you that you can write your own program with very little investment of time, effort

(Continued on Page 26)

Advanced CP/M

Zex 5.0—The machine and the Language

by Bridger Mitchell

The Z-System batch processor is ZEX—the Z-System EXecutive input processor. In TCJ #38 I focused on redesigning the ZEX language used to specify input in the form of a script of lines in a file, or entered interactively from the console.

My goal was to make a ZEX script easy to write and read—by using keywords rather than reserved punctuation symbols as directives—and to distinguish clearly between input for the command processor and input for a program. Implementation, testing, and excellent suggestions from Carson Wilson, Jay Sage, Howard Goldstein, Cam Cotrill and Rick Charnes have improved and streamlined the draft specification I put forward then. A summary of the nearly-final specification is found later in this column.

At deadline time for this column the new ZEX—dubbed ZEX 5.0—appears to be nearing the end of its testing cycle. It will be available on Z-Nodes about the time this issue reaches you and included on future NZ-COM and Z3PLUS release disks.

ZEX will doubtless continue to evolve, and it should. It is an outstanding example of the cumulative contributions of the Z-System community. One of my goals in rewriting earlier versions by Rick Conn, Joe Wright and Jay Sage was to make the functional routines more modular and the logic more transparent, so that further experimentation and new extensions would become easier.

Nevertheless, ZEX is intrinsically a complex and highly interdependent software machine. In this column I will attempt to give you an overview of its more important components. The source code, which is too extensive to even excerpt here, will also be available on Z-Nodes for closer study. I do ask, however, that you not release any modified version without coordinating with me and Jay Sage, the ZSIG Librarian.

Preprocessing and Loading.

ZEX divides naturally into two functional components. The first preprocesses the ZEX script and loads it and the ZEX monitor into high memory. The second is the resident ZEX monitor itself which interprets the script as the command processor and programs request console input.

The ZEX loader reads a script written in the ZEX language (from a file, or interactively) and *compiles* it into a compact, internal format that is stored in resident memory in a buffer within the ZEX monitor. (The format consists of substituting 8-bit codes for each of the ZEX directives and carrying along the command lines and program input as entered.) The loader code then relocates the monitor module to high memory, and links the BIOS and BDOS intercepts into the monitor.

Compilation involves parsing input directives such as |UNTIL x| into their internal format, converting the ends of each script line into a <CR>, a <CR><LF>, or neither, performing textual substitution for dummy parameters such as $1, and marking each line as input for the command processor or program.

Two versions of the monitor module, one for CP/M 2.2 and one CP/M Plus, are contained within the ZEX.COM image as two blocks of PRL (page-relocatable) code. ZEX detects which system is running, and relocates and installs the appropriate monitor. Thus, a single ZEX.COM file serves for all Z-Systems.

Implementing the ZEX Monitor

The active portion of ZEX is an RSX—a resident system extension to the Z-System DOS and BIOS that is located in memory below the command processor (and any other pre-existing RSXs). It intercepts the three BIOS console input and output functions—constat, conin, and conout. In addition, as an RSX, it must intercept the BIOS warmboot function to prevent a complete warmboot that would otherwise reload the command processor and change the top of user memory (TPA) address at location 0006.

The RSX design of ZEX 5.0 follows the standards I set out for CP/M 2.2 RSXs in TCJ #34. It begins with a Plu*Perfect Systems RSX header containing vectors to the BDOS entry, the BIOS warmboot intercept, and the RSX removal routine, plus values of the original warmboot address, the lowest address in the RSX that must remain protected in memory, and a pointer to a nul-terminated name of the RSX ("ZEX 5.x").

Using what I hope is now the standard method of coding a (CP/M 2.2) RSX gains several advantages. A ZEX script is able load (run) another standard RSX, and can also remove it, because the interface between the two is fully defined. ZEX can also be used when a standard RSX is already present. And finally, the standardized header and intercept routines largely prepackage the intricacies of coding the linkages to the BIOS and BDOS.

The CP/M Plus version of ZEX 5.0 also begins with a standardized header—the one defined by Digital Research for CP/M Plus RSXs. Because ZEX intercepts BIOS functions that will be called by the BDOS in banked memory it must be loaded (and removed) with careful attention to the secondary jump table used by the banked BDOS to access BIOS functions.

In addition to providing the standard memory-protection func-

---

*Bridger Mitchell is a co-founder of Plu*Perfect Systems. He's the author of the widely used DateStamper (an automatic, portable file time stamping system for CP/M 2.2); Backgrounder (for Kaypros); BackGrounder ii, a windowing task-switching system for Z80 CP/M 2.2 systems; JetFind, a high-speed string-search utility; DosDisk, an MS-DOS disk emulator that lets CP/M systems use pc disks without file copying; and most recently Z3PLUS, the ZCPR version 3.4 system for CP/M Plus computers.*

*Bridger can be reached at Plu*Perfect Systems, 410 23rd St., Santa Monica CA 90402, or at (213)-393-6105 (evenings).*

tion, the ZEX warmboot routine checks to see whether ZEX should be removed from memory. Usually, if the ZEX script has been exhausted (or ZEX has been canceled by an application, such as an error handler) it should be removed no later than the next warmboot. But if an RSX has been loaded below ZEX, its removal must be postponed until that lower RSX is no longer present.

When its time has come, the removal routine restores each intercepted BIOS jump vector to its status at the time ZEX was loaded. (In the CP/M Plus version this also requires restoring the secondary jumps used by the banked BDOS). It also restores the original status of the Z-System quiet flag.

The Console Intercept Functions

The heart of ZEX is the logic of the three key RSX routines—constat, conin, and conout. I'll summarize the major features.

In its simplest form, ZEX *redirects* console input to the command processor—instead of having command lines come from the keyboard, they come instead from a script. In this form, ZEX is effectively just an in-memory form of the original SUBMIT program for CP/M, the most basic form of batch processing.

This degree of redirection is straightforward. As an RSX, you intercept just the BIOS conin function, and each time it is called by the command processor you return the next character in the script.

Script Input to a Program

The first major advance of ZEX over vanilla SUBMIT is that it can also redirect the console input of a *program*. This means that what is ordinarily an interactive application can be scripted to run on auto-pilot.

Unfortunately, achieving this step is fraught with tangles. ZEX must now intercept BIOS constat as well as conin and intelligently indicate to an application whether a character is waiting to be read (console status true) or not. At first, it seems "obvious" that the ZEX constat function should always return "true", so long as there is another script character to be returned.

But this turns out to be wrong for a number of major applications that attempt to "look ahead" to see if the user is typing faster than they are processing input. Depending on the application, such programs may repeatedly call conin to remove the characters, stop displaying any output, or even overrun their buffers (a bug). Consequently, ZEX's constat routine must sometimes pretend that no character is yet ready to be read.

Providing script input to an application as well as to the command processor creates another major complication--how to keep the input destined for each clearly separated. Prior to ZEX 5.0, the script consisted of an undifferentiated stream of bytes--command line, program input, next command line, etc. All too frequently, an extra (or a missing) character caused the rest of an extensive script to crumble into gibberish, and even wrought disaster if the wrong character answered an interactive reply such as "Delete all?".

By extending a feature in CP/M Plus's SUBMIT, ZEX 5.0 differentiates script input to a program from input to the command processor. Each line of program input begins with the '<' character; any other line is command processor input.

In order for this to work, ZEX must know at all times whether the command processor or an application is calling conin. This information is provided in a one-bit message controlled by ZCPR3--when the command processor is requesting input (or sending console output) the bit is set, when it transfers control to a program (including an RCP) the bit is cleared.

Adapting to Run-Time Conditions

ZEX's second major advance over SUBMIT is a considerable degree of flexibility in tailoring the script to *run-time* conditions. The script can provide input to an application up to a certain point, then switch to console input, and later resume with additional script. Or, the program can obtain input from the console until the program sends a specific output message, then obtain further input from the script. Portions of the script can be conditionally executed--for example, only if the ZCPR3 command-flow state is true (or false).

Control of Output

Finally, ZEX provides flexible control over console *output*. Output from the command processor, or also from a program, can be turned off, so that a complex sequence of operations runs "silently," without any distracting (or revealing) screen displays. And the script may at any point emit its own messages to the screen. In effect, it is possible to completely "repackage" the view the user has of a sequence of applications.

Implementing these advanced features requires a quite complex web of conditional routines and state flags in the constat, conin, and conout intercept routines. ZEX keeps track of the current state (command processor, application), the previous state, and the current input mode for each state (console, script, waiting-for-keyboard-character, watching-for-output-string).

ZEX attempts to keep the script synchronized with the current state. If there is no script input for the current state, ZEX switches to console mode. So if the script, for example, provides the answers to the first of an application's two questions, the keyboard will be used for the second. When a state transition occurs (from ccp to application, or the reverse), ZEX will discard any input that might remain for the just-finished state. If there were unread script input for an application when it terminated, ZEX would move ahead to the next command-line input.

Hopefully, synchronization, coupled with the discipline of clearly designating each line of input for either the command processor or the application, will greatly reduce the difficulty of developing and running complex scripts.

Thinking about ZEX

ZEX is an unusually complex RSX, and it's not always easy for even the developer to get his head clearly around everything that is going on! It's helpful, however, when planning a script, to keep in mind just *how* ZEX interacts with the rest of the system.

A ZEX script is a linear stream of bytes--console input supplemented by directives and console output messages. Since ZEX intercepts only constat, conin, conout, and warmboot, it is only when one of those functions is called that ZEX can have any effect. The next byte in the script is normally processed only when conin is called. (An exceptions: if a |WATCHFOR| directive is the first element of program input, it becomes active as soon as the application calls conout or conin.) Thus, most directives do not actually take effect when a program begins (or when the command processor is reentered), but when it requests the first console character.

Examples of ZEX scripts

Time and space don't allow me to include actual scripts, but you will see them showing up as files with filetype .ZEX on the Z-Nodes. To get familiar with ZEX it's handy to use it in interactive mode, typing in a couple of test lines and watching what happens. The "ZEX //<cr>" command will remind you of the directives, and a "GO<cr>" will immediately reinvoke ZEX.

More extensive applications should be composed as a file, including lots of comments that make the scripts self-documenting. I would enjoy receiving any favorites that you work up. •

A Reference Guide to the ZEX Language

The ZEX script consists of lines of ASCII text, each terminated by a <CR><LF> pair. You can create the script with a text editor in ASCII (non-document) mode. For short, one-off uses, invoke ZEX with the command "ZEX<cr>" and type the lines into ZEX when prompted.

A number of reserved words, called *directives*, control the various features. Each directive begins and ends with the verticule character '|'. The directives may be entered in upper, lower, or mixed case; I use upper case here to make them stand out. All script input that is to be sent to a program begins with a '<' character in the first column; all other lines are sent to the command processor or, when specifically directed, are messages sent directly to the console output.

Command-processor input:

- is any line of the script that doesn't begin with '<'
- is case-independent.
- spaces and tabs at the beginning of a line are ignored
- The end of a script line is the end of one command line. Use the |JOIN| directive at the end of a script line to continue the same command line on a second script line. (The <LF> is always discarded).
- all white space immediately preceding a |JOIN|, and all characters on the line following |JOIN|, including <CR>, are discarded.
- use |SPACE| to precede a command with a space, or to insert a space after a command and before a comment.
- begin each command on a new script line, optionally preceded or followed by white space. Although multiple commands may appear on the same line, each separated by a semicolon, this should be avoided to reduce the chance that the Z-System multiple command line buffer will overflow.

Program input:

- is normally obtained from the console.
- begin each script line of program input with a '<' in the first column.
- input is case-sensitive.
- data from the script includes the <CR> (but not the <LF>) at the end of a script line. To omit the <CR>, use the |JOIN| directive.
- use |LF| for linefeed, |CRLF| for carriage-return-linefeed, |TAB| for tab, |DEL| for 7fh, |NUL| for 00h.
- if the program requests more input than is supplied in the script, the remaining input is obtained from the console
- use |WATCHFOR string| to take further input from the console, until the program sends "string" to the console output, then resume input from the script

Both:

- use |SAY| to begin text to be printed on the console, and |END SAY| to terminate that text. Within a |SAY| directive that extends over more than one line of script, the <CR> *and* the <LF> are both output to the console. Use |JOIN| if this is not desired.
- use |UNTIL X| to take further input from the console, until a keyboard 'X' is entered. The 'X' character may be any character; pick one that won't be needed in entering console input.
- use |UNTIL| to take further input from the console, until a keyboard <CR> is entered.

Script Comments

A double semicolon ";;" designates the beginning of a comment. The two semicolons, any immediately-preceding white space, and all text up to the <CR> of that line of script are ignored.

A left brace '{' in the first column designates the beginning of a comment field; all text, on any number of lines, is ignored up to the first right brace '}'.

Other Directives

Within a directive, a SPACE character is optional. Thus, |IF TRUE| and |IFTRUE| have the identical effect.

Conditional script directives:

```
|IF TRUE|     do following script if command flow state is true
|IF FALSE|    do following script if flow state is false
|END IF|      end conditional portion of script
|IF TRUE| <a> |ELSE| <b> |END IF|   do <a> if true, do <b> if false
```

Miscellaneous directives:

```
|RING|          ring console bell
|RING WAIT|     ring bell and wait until a <CR> is pressed
|WAIT|          wait until a <CR> is pressed
|AGAIN|         repeat the entire ZEX script
|ABORT|         terminate the script
```

Directives that control console output:

```
|CCPCMD|    / |CCPCMD OFF|     Show/ Don't show: CCP command output
|ZEXCMD|    / |ZEXCMD OFF|     Show/ Don't show: the ZEX command prompt
|FALSECMD|  / |FALSECMD OFF|   Show/ Don't show: commands in false flow state
|SILENCE|   / |SILENCE OFF|    Suppress/ resume: console output
|QUIET|     / |QUIET OFF|      Set/ Unset:       ZCPR quiet flag
```

For each of these console-output directive words (...) the following synonyms are recognized: |...| or |... ON| or |... YES|, and |... NO| or |... OFF| or |END ...|.

The |SILENCE| directive suppresses all console output except that in a |SAY| message.

Special character directives:

```
|CR|        carriage return
|LF|        linefeed
|CR LF|     carriage return, line feed
|TAB|       horizontal tab
|NUL|       binary null
```

```
|SPACE|      space character
|DEL|        delete (7Fh) character
```

Parameters

ZEX (like SUBMIT) provides for formal parameters designated $0 $1 ... $9. When ZEX is started with a command line such as:

```
A> ZEX SCRIPT1 ARG1 ARG2 ARG3
```

then ZEX reads and compiles the SCRIPT1.ZEX file. In the script, any "$0" will be replaced by "SCRIPT1", any "$1" is replaced by the "first" argument "ARG1", etc.

The script may define "default parameters" for the values $1 ... $9. To do so, enter the three characters "^$n" followed (with no space) by the nth default parameter. When ZEX encounters a formal parameter in the script, it substitutes the command-line parameter, if there was one on the command line, and otherwise the corresponding default parameter, if it was defined.

Control characters

You enter a control character into the script by entering a caret '^' followed by the control-character letter or symbol. For example, "^A" will enter a Control-A (01 hex). Control-characters may be entered in upper or lower case.

Quotation

ZEX uses a number of characters in special ways: dollar-sign, caret, verticule, left and right curly braces, less-than sign, semicolon, (space, and carriage-return). Sometimes we might want to include these characters as ordinary input, or as output in a screen message. For this, ZEX uses '$' as the *quotation character*. (This is also called the *escape* character, because it allows one to escape from the meaning reserved for a special character.) "Quotation" means that the next character is to be taken literally; I use this term to avoid confusion with the control code 1B hex generated by the *escape key*.

If '$' is followed by any character other than the digits from '0' to '9', that character is taken literally. Thus, if we want a caret in the text and not a control character, we use '$^'. If we want a '<' in the first column of a line that is for the command processor and not for program input, then we use '$<' there instead. And don't forget that if we want a '$' in our script, we must use '$$'. There are some cases, like '$a', where the '$' is not necessary, but it can always be used.

To pass a ZEX directive to a program, or the command processor, use the quotation character with the verticule. For example, to echo the string "|RING|", the zex script should be:

```
echo $|RING$|
```

WordTech's dBXL

(Continued from page 22)

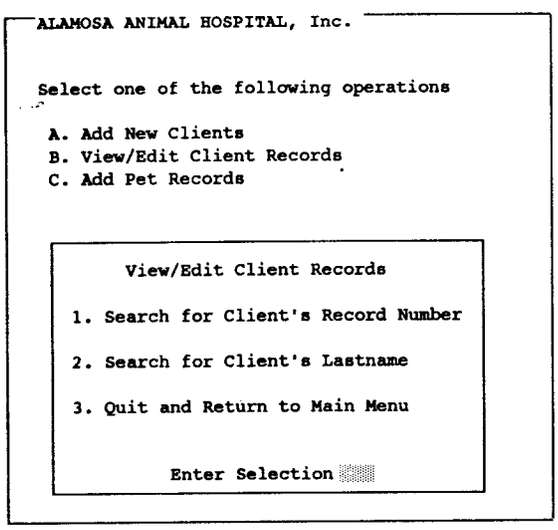

ALAMOSA ANIMAL HOSPITAL, Inc.
Select one of the following operations
A. Add New Clients
B. View/Edit Client Records
C. Add Pet Records
View/Edit Client Records
1. Search for Client's Record Number
2. Search for Client's Lastname
3. Quit and Return to Main Menu
Enter Selection

Figure 9: Demonstration of windowing capability for secondary screens.

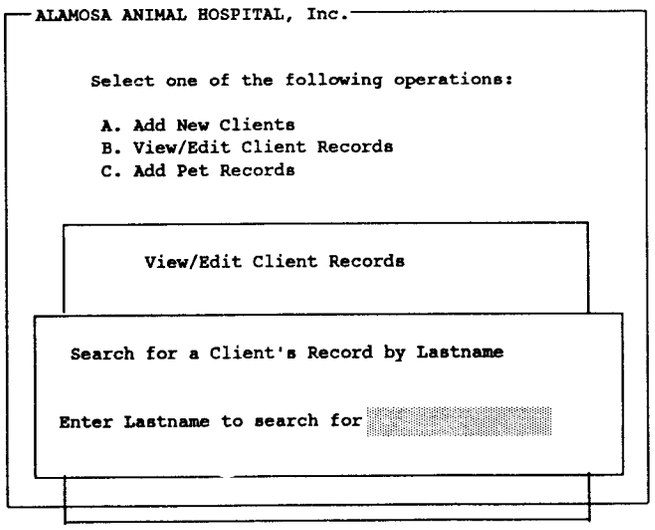

ALAMOSA ANIMAL HOSPITAL, Inc.
Select one of the following operations:
A. Add New Clients
B. View/Edit Client Records
C. Add Pet Records
View/Edit Client Records
Search for a Client's Record by Lastname
Enter Lastname to search for

Figure 10: As many as 99 windows can be opened on the screen at one time

or money. You will develop a sense of pride at your finished product, and, who knows, you may develop a marketable program that you can sell to other individuals in your profession.

If you write to WordTech, they will send you a demo disk at no charge. I suggest you do that. Their address is PO Box 1747, Orinda, CA 94563. Their phone is (415)254-0900. I might also suggest that you look over a copy of "Understanding & Using dBASE III Plus" by Rob Krumm. •

Programming for Performance

Advanced Assembly Language Techniques

by Lee A. Hart

Truly efficient software has an intimate, almost incestuous relationship with its hardware. They merge so thoroughly as to become inseparable; neither makes any sense without the other.

This requires that you, the programmer, must TOTALLY understand the hardware. I cannot stress this point too strongly. The strengths and weaknesses of the hardware influence program structure at every level, not just the low-level drivers. A system with weak disk I/O will be slow and unresponsive if your program relies on overlays. A shallow keyboard buffer requires frequent checks to avoid missing keys. The characteristics of the console device determines your whole approach to data displays. If you try to hide from these limitations in a high-level language, your program will work as if it were written in BASIC 101. Let's consider some actual case histories of what can be gained by paying attention to the hardware.

CASE #1

A customer needed a faster way to transfer data between two computers. He had been using a serial port at 9600 baud but complained that it was too slow and tied up the computer's serial port. Hardware mods were ruled out.

After study, I found that each computer had unused handshake lines in its RS-232 port. A special "Y" cable was built to cross-connect two of these lines, providing one bit of serial I/O in each direction. A "software UART" program was then written to transfer data between the two machines. This worked to about 30K bits per second before timing dither (due to interrupts, memory refresh, etc.) caused errors.

The serial port's UART could be programmed to generate an interrupt when the handshake line went low. Therefore, an interrupt-driven protocol with handshaking was devised. A '0' was sent by pulling the output low until the other computer echoed the low on its output. A '1' was sent by pulsing the output low and immediately back high and waiting until the other system echoed it. The data rate increased to over 100K bits per second, and transfers were now unaffected by disk I/O, keyboard activity, etc.

CASE 2

The firmware for a CRT terminal was to be upgraded to run 38400 bits per second without handshaking. Now, 38400 bps is fast, only 260 microseconds per character (about 75 instructions for a 3 MHz Z80).

The slowest routines need the most attention. For example, clear-line was accomplished by moving the stack pointer to the end of the line and executing 36 PUSH HL instructions. The interrupt handler needed a 4-level stack, so the last 8 bytes were cleared normally. Clear-screen used 25 iterations of clear-line.

This still isn't fast enough to complete every ESC sequence before the next one is received. This calls for an interrupt-driven system. Each character received generates an interrupt. The interrupt handler pushes the character into one end of a FIFO (First-In-First-Out) buffer in memory. The main program pops characters out the other end and processes them. The FIFO fills while we process slow commands like clear-screen and empties back out during fast commands.

But what if some idiot sends a long string of slow commands (like 100 clear-screens in a row)? The FIFO would eventually overflow, and data would be lost. I prevented this with "look-ahead" logic. When the interrupt handler spots a clear-screen command, it sets a flag so MAIN expects it. MAIN can then ignore unnecessary commands (no sense altering a screen that's about to be cleared).

Scrolling is one of the most difficult actions. The obvious algorithm is to block move lines 2-24 up 1, then clear line 24. But that's what IBM did on the PC, and we all know how well that worked. So examine the 6845 CRT controller. The Start-Address register holds the address of the first character on the screen, the one displayed in the top left corner. If we add 80 to it, line 2 instantly becomes the top line, and we've scrolled the whole screen up a line. All that remains is to clear the 80 bytes that form the new 24th line, for which we have a fast routine.

Each scroll moves the start address up another 80 bytes. This obviously can't go on indefinitely, so the original program spent a great deal of time checking for overflow outside its 2K block of screen RAM (F800-FFFF). For instance, the old code read as shown in Figure 1.

But is this really necessary? The schematic revealed that the 2K RAM was partially decoded and actually occupied 16K in the Z80's address space (C000-FFFF). It's far easier to insure that an address lies within this range as shown in Figure 2.

CASE #3

Fast Disk I/O. Way back in 8 B.C. (eight years Before Clones) I had an S-100 system. Its 8080 CPU blazed along at 1.843 MHz, through 32K of RAM spread over half a dozen furnace boards. Two Shugart SA-801R single-sided 8" drives provided disk storage, with CP/M 1.4 tying it all together. That old war horse and I fought many battles together, until it finally died the Death-of-1000-Intermittents.

Many of its "features" I'd rather forget, but it had one outstanding attribute: the fastest floppies I've ever seen. Warm boots were done before your fingers were off the keys; Wordstar loaded in under a second; PIP copied files at 10K bytes/sec. All without a fast CPU, DMA, vectored interrupts, or even a disk controller IC. The "controller" was just a bunch of TTL chips implementing a parallel port, an 8-bit shift register, and a CRC checkcode generator. The real work was done by the CPU, byte-banging out the IBM 3740

SD/DD format in software.

How good was it? An 8" disk spins at 360 rpm, or 6 revs/sec. Each track held 6.5K (26 double-density sectors of 256 bytes each). That makes the theoretical maximum transfer rate 6.5K x 6 = 39K bytes/sec. It actually achieved 50% of this, or 20K bytes/sec.

Few modern micros come anywhere near this level of performance. The Kaypro I wrote this article on creeps through the disk at 4K/sec. My PC clone is closer, at 12K/sec. The problem is that the CPU spends most of its time in wait loops; waiting for the drive motor to start, for the head to load, for an index hole, for a certain sector to come around on the disk. The capabilities of fast CPUs, elaborate interrupt systems, DMA, and fancy disk controllers are thrown away by crude software.

The CPU has better things to do. If the disk isn't ready when an application program needs it, the BIOS should start the task, save the data in a buffer, and set up an interrupt to finish the task later when the disk is REALLY ready. The time lost to wait loops is thus reclaimed to run your application programs.

That's how my antique worked. The BIOS maintained a track buffer in RAM. The first read from a particular track moved the head to the desired track and read the whole thing into the buffer. Further reads from that track simply came from RAM, taking virtually no time at all.

Similarly, writes to a sector on the current track just put data in the buffer and marked it as changed. The actual write was performed later, when a new track was selected for read/write, or just before the drive timed out from a lack of disk activity.

Physical track reads/writes were fast as well. The key was to simply begin wherever the head was. After seeking to the desired track, it read the ID# of each sector encountered and transferred it to/from the appropriate place in the RAM buffer. No need to find the index hole, wait for a particular sector ID#, or worry about interleave; one revolution got it all.

Such a system must be implemented carefully. CP/M does not expect delayed error messages, which can produce some odd results. For instance, a BDOS read error might be reported when the real cause was a write error in flushing the previous track buffer to disk. Also, modern drives do not have door locks to prevent disk removal if unwritten data remains in the track buffer.

The main factor limiting my S-100 system's performance was the slow CPU and lack of DMA. A double-density 8" disk has a peak data transfer rate of 500K bits/sec, which only allows 16 microseconds between bytes. This required polled I/O where the CPU was 100% devoted to the disk during actual reads/writes.

5.25" disks have a slower maximum transfer rate, but modern hardware has advantages that can make up for it. A normal 5.25" disk spins at 300 rpm, or 5 revs/sec. Assuming 9 sectors of 512 bytes per track, the maximum transfer rate is 22.5K bytes/sec. The peak data rate is 250K bits/sec, or 32 microseconds per byte. This is slow enough for a 4 MHz Z80 to (barely) handle it on an interrupt basis. Figure 3 shows an interrupt handler to read 256 bytes from a disk controller chip at 32 microseconds max. per byte.

Figure 1:

```
     ld   (hl),a    ; put character on screen
     inc  hl        ; advance to next
     ld   a,h       ; get new address
     or   0F8h      ; if overflow to 0000,
     ld   h,a       ;   force it to F800-FFFF
```

Figure 2:

```
     ld   (hl),a    ; put character on screen
     res  6,h       ; insure we don't wrap to 0000
     inc  hl        ; advance to next
```

Figure 3:

```
T-states
   23                       ; time to finish longest instruction
   13                       ; Z80 interrupt mode 0 or 1 response
   11 int:  push af         ; save registers used
   11       in   a,(data)   ; read data byte from disk controller
   13 next: ld   (buffer),a ; store it in buffer (a variable)
   13       ld   a,(next+1) ; get buffer address
    4       inc  a          ;   increment
   13       ld   (next+1),a ;   save for next time
    7       jr   nz,done    ; if end of page, done
   10       pop  af         ;   else restore registers
   10       ret             ;   and return
  ----
  128 T-states max = 32 microseconds with a 4 MHz Z80
```

Figure 4:

```
T-states
   23                       ; time to finish longest instruction
   19                       ; Z80 interrupt mode 2 response time
   11 int:  push af         ; save A and flags
   11       in   a,(data)   ; read 1st byte from disk controller
   11       push hl         ; save HL
   10 next: ld   hl,buffer  ; get buffer address (a variable)
    7       ld   (hl),a     ; store byte in buffer
    6       inc  hl         ; advance buffer pointer
    6       inc  hl         ;   for next interrupt
   16       ld   (next+1),hl;   & store it
    6       dec  hl         ;   point to current address
11+11 check:in   a,(status) ; check disk controller status
 4+ 4       rra             ; if not busy (bit 0=1),
 7+ 7       jr   nc,done    ;   then we're done
 4+ 4       rra             ; if next byte not ready (bit 1=0),
12+ 7       jr   nc,check   ;   then loop until it is
   11       in   a,(data)   ; get 2nd byte from disk controller
    7       ld   (hl),a     ;   & store it in buffer
   10       pop  hl         ; restore registers
   10       pop  af
   14       reti             ; return
  ----
  188 or 226 T-states (for 1 or 2 passes through status loop)
```

But this routine barely squeaks by. It can't use interrupt mode 2 (which adds 6 T-states to the response time) or signal Z80 peripherals that the interrupt is complete with an RETI (which adds 4 T-states). It's limited to a 256-byte sector. Worse, some disk controller chips need processing time of their own. The popular Western Digital FD179x series only allows 27.5 microseconds for each byte.

So we have to get clever again. The example in Figure 4 reads pairs of bytes, the first on an interrupt and the second by polled I/O. This improves performance to allow interrupt mode 2, larger sector sizes, and the slow response time of a FD179x chip.

This routine reads bytes from the controller chip within 17.75 microseconds worst-case. Interrupt overhead averages 80% for a 4 MHz Z80, leaving 20% for the main program execution. The peculiar way of incrementing the address pointer minimizes the worst-

case delay from an interrupt or status flag change until the byte is read. We want to maximize the chance that the second character is ready the first time the status is checked.

Why improve your disk system? Because, as a practical matter, there's more to be gained by improving it than any other change you could make. It's disk I/O that sets the pace, not CPU speed or memory size. Users almost never wait on CPU speed; it's the disk that keeps you twiddling your thumbs with the keyboard ignored, the screen frozen, and the disk drive emitting Bronx cheers. Put a Commodore 64's tinkertoy disk system on an AT-clone, and you'd have high-priced junk that only a masochist would use. Conversely, the AT's DMA-based disk I/O would transform a C64 into a fire-breathing dragon that would eat its competition alive.

Algorithms

When a hardware engineer sits down to design a circuit, he doesn't begin with a blank sheet of paper. He has a vast library of textbooks, data sheets, and catalogs of standard circuits to choose from. Most of the task is simply connecting off-the-shelf components into one of these standard configurations, modifying them as necessary to satisfy any unique requirements.

Algorithms are to programmers what IC chips are to hardware designers. Just as the engineer builds a library of standard parts and circuits, every programmer must continually build his own algorithm collection. Whether it's a shoebox full of magazine clippings or a carefully indexed series of notebooks, start NOW.

Programming textbooks tend to concentrate on traditional computer algorithms for floating-point math, transcendental functions, and sorting routines. The old standby is Knuth's "The Art of Programming". Hamming's "Numerical Methods for Scientists and Engineers" explains the basics of iterative calculations. "Digital Computation and Numerical Methods" by Southworth and Deeleeuw provides detailed flowcharts and sample code as well.

Magazines are a great source and tend to be more down-to-earth and closer to the state of the art. Read carefully! Good algorithms may be expressed in BASIC listings, assembly code for some obscure processor, pocket calculator key sequences, and even disguised as circuit diagrams. Professional journals like EDN or Computer Design are often better than the popular magazines, which have pretty much abandoned education in favor of marketing. Especially check out back issues. The cruder the hardware, the trickier the algorithms had to be to make up for it.

Manufacturers' technical literature is a gold mine. Get the manufacturers' own manuals, not some boiled-down paperback from the bookstore. They won't be models of clarity but are full of hidden gold. Read everything, hardware and software manuals, data sheets, application notes, etc.

User groups are the traditional source of solutions to specific problems. Even better, they provide actual implementations in printed listings, on disk, or even by modem.

Don't waste time reinventing the wheel. Learn from others what works, and what doesn't. Some of the best (and worst) algorithms I know were found by disassembling existing programs. And once you find a good algorithm, recycle it. That clever sort routine for an antique 8008 may be the foundation of the fastest '386 sort yet!

Conclusion

These techniques are not new; in fact old-timers will recognize many of them from the early days of computing when hardware limitations were more severe. However, they have fallen into disuse. A whole generation of programmers has been taught that such techniques have no place in modern structured programming.

The theory goes something like this: Programs written in a high-level language are faster and easier to write, debug, document, and maintain. Memory and speed are viewed as infinite resources, so the performance loss is unimportant. Programs should be totally generic; it is the compiler's or run-time library's job to worry about the hardware interface.

These rules make sense in a mainframe environment, where the hardware resources are truly awesome, and teams of programmers spend years working on one application. But they impose severe penalties on a microcomputer system. The user must pay for the programmer's luxuries with higher hardware cost and lackluster performance.

It's easy to forget that "microcomputer" literally means "one millionth of a computer". Microprocessors make abysmally bad CPUs. Build a computer with one, and you'll wind up needing $5000 worth of memory and peripherals to support a $5 CPU chip.

But micros make superlative controllers. That's what they were designed for, and what they do best. A single microcomputer can replace dozens of boards and hundreds of ICs with as little as a single chip. That's why 90% of all microprocessors go into non-computer uses: calculators, auto emission controls, home entertainment equipment, industrial controls, and the like. Of 30 million Z80s sold last year, fewer than 1 million went into computers.

Programming a controller is different than a computer. Most applications demand real-time multi-tasking capabilities, and there is never enough speed or memory. Inputs and outputs are physical hardware devices, not abstract data structures, so the code must inevitably be hardware-dependent. Computer languages are just not cut out for this sort of thing.

The question is not, "How do I write a computer program to handle this data?" Instead, you should ask yourself, "How must I manipulate this hardware to do the job?" The techniques in this article may be out of place in the bureaucracy of a large computer but are right at home in the wild-west world of a microcomputer.

Lest you think this has nothing to do with a "real" computer like your PC clone, consider this. Instead of a '286 with 1 meg of memory, suppose it contained ten Z80 controller boards, each with 64K of memory and a fast network to tie them together. Each Z80 handles a different device: keyboard, screen, printer, modem, and one for each disk. The rest are free to run application programs, several at a time!

Suppose you're doing word processing on this system. The keyboard does spelling correction on data entry. The screen Z80 displays your text in bit-mapped fonts to match the printer's Z80, which is simultaneously printing a file. The Z80 running the word processor itself suffers no annoying pauses or hesitation, since disk I/O is handled instantaneously via each drive's Z80 track buffer. Meanwhile, the modem's Z80 is downloading a file while another assembles a program. Pop-up utilities are ready and waiting in still other Z80s in case they're needed.

Such a system would clearly have half the hardware cost of a PC, yet would outperform it by a wide margin. True multi-tasking becomes child's play with multiple processors. More processors can be readily added for even higher performance, or removed to save cost (or continue operation while waiting for a replacement).

If the computer scientists really want to further the micro-revolution, they should stop trying to force antiquated mainframe languages onto micros and concentrate on developing tools to maximize the use of micros as they are! •

Programming Input/Output With C

Part 1: Keyboard and Screen Functions

by Clem Pepper

C is unique among programming languages in not providing for input and output as an integral part of itself. Obviously, the means to communicate are provided somewhere. The "somewhere" are library functions to be included for compilation with our programs. These in general are provided as a library with the compiler at the time of purchase. Library files are not ASCII so we are unable to read them unless we have obtained the source code.

As the forthcoming ANSI standard wends its way toward final acceptance the majority of vendors are adhering to its functional specifications. The standardized library functions are quite powerful in paving the way for program communication between the keyboard, the screen and external devices. Adherence to the standard assures input/output portability.

The Stream Concept

I first encountered the stream concept about four years ago when I obtained TOOLWORKS C compiler, provided by The Software Toolworks. Simply stated, the stream is the flow of data characters organized as a series of bytes. There are two kinds of streams: text and binary. The focus of this article is on text, but with some few exceptions the requirements will apply to binary files as well.

Stream variables are of the type FILE. FILE is a structure defined as a type by typdef. FILE * is a pointer to structures defined in the library file <stdio.h>. Note that "h" and "c" files are written and compiled identically. However the "h" extension identifies these as header files to be included at compile time. The header files make calls to other library functions. Although header files are embedded within the library code, we are liberty to write header files of our own to take advantage of frequently used functions of our creation.

We typically put "#include <stdio.h>" at the top of our programs. "stdio" means standard input/output. stdio.h is in the library provided with our compiler.

The keyboard is the default device for stream input; our monitor screen the default for stream output. Structures in stdio.h direct the stream flow. A third structure, stderr, is initialized to a screen output stream to receive error messages and unexpected output from a program. The declarations for these are:

```
extern FILE *stdin;        /* from the keyboard */
extern FILE *stdout;       /* to the screen     */
extern FILE *stderr;       /* to the screen     */
```

stdin and stdout may be re-directed to disk files, a printer or other peripheral. stderr is always directed to the screen.

Keyboard Input

Two basic situations complicate reading the MS DOS keyboard. The first is that two kinds of keys are present, ASCII and non-ASCII. The ASCII keys are those we associate with a typewriter, plus some others such as ESC and Back Space and CTRL functions. The non-ASCII are the remaining: notably the ten (or 12) function keys and the keypad. These may be in any one of four modes: unshifted, shifted, control (CTRL) or alternate (ALT). The description non-ASCII is used as these keys provide a two part number having a numerical value unrelated to any standard ASCII function.

Decimal scan code values for the non-ASCII keys are listed in Table 1. The "scan code" is a number identifying the pressed key. DOS replaces the scan code with its ASCII equivalent where such exists. The ASCII values are easily read by our program. The non-

KEY	UNSHIFTED	SHIFTED	CONTROL	ALTERNATE	FUNCTION
F1	0 59	0 84	0 94	0 104	Function Key 1
F2	0 60	0 85	0 95	0 105	Function Key 2
F3	0 61	0 86	0 96	0 106	Function Key 3
F4	0 62	0 87	0 97	0 107	Function Key 4
F5	0 63	0 88	0 98	0 108	Function Key 5
F6	0 64	0 89	0 99	0 109	Function Key 6
F7	0 65	0 90	0 100	0 110	Function Key 7
F8	0 66	0 91	0 101	0 111	Function Key 8
F9	0 67	0 92	0 102	0 112	Function Key 9
F10	0 68	0 93	0 103	0 113	Function Key 10
./DEL	0 83	46 83	-----	-	Delete Character
0/INS	0 82	48 82	-----	0	Key Pad/Insert Mode
1/End	0 79	49 79	0 117	1	Key Pad/End of Line/File
2/Down	0 80	50 80	-----	2	Key Pad/Cursor Down
3/PgDn	0 81	51 81	0 118	3	Key Pad/Page Down
4/Left	0 75	52 75	0 115	4	Key Pad/Cursor Left
5	-----	53 76	-----	5	Key Pad
6	0 77	54 77	0 116	6	Key Pad/Cursor Right
7/HOME	0 71	55 71	0 119	7	Key Pad/Cursor Home
8/Up	0 72	56 72	-----	8	Key Pad/Cursor Up
9/PgUp	0 73	57 73	0 132	9	Key Pad/Page Up
*/PrtSc	42 55	------	0 114	-	*/Print Screen

Table 1. Special Function Key Scan Codes.

ASCII are another matter.

It is necessary to see just how keyboard scan codes are dealt with by MS DOS. Keyboard Input/Output is dealt with by one of three operation codes, 0, 1 or 2, under interrupt 16H (INT 16H). To perform a keyboard function a program loads the operation code into register AH and then executes a INT 16H instruction.

Operation Code 0 gets the character from the keyboard. When a character is typed its ASCII code will be entered in register AL. The scan code will be entered in register AH. An illustrative function for reading non-ASCII keys is:

```
/* == read function or other non-ASCII key == */
int rd_nonasky()
{
   union REGS regs; /* dos.h union */
    regs.h.ah = 0; /* Operation code, replaced by ASCII */
    regs.h.al = 0; /* Operation code, replaced by scan
                      code */
      int 86(SFKEY, &regs,&regs);
    /* transfer register values to memory */
      asc = regs.h.ah;  /* ASCII code */
      scn = regs.h.al;  /* scan code */
```

In application, variables asc and scn are declared as integers from the calling program, or declared as globals. (Listing 8 is a complete program using this function.)

Operation Code 1 determines if a key is waiting to be input from the keyboard buffer. If the Zero flag is set, no character was present. If the flag is clear the ASCII code will be in AH; the scan code in AL as above. The keycode is retained in the buffer; the next function to code 0 will return the key value.

Operation Code 2 returns the keyboard's shift status. The value returned in register AL is bit-mapped as follows:

```
0   Right SHIFT key pressed.
1   Left SHIFT key pressed.
2   CTRL key pressed.
3   ALT-SHIFT pressed.
4   Scroll-lock state; 1 returned if active.
5   Numeric-lock state; 1 returned if active.
6   Caps-lock state; 1 returned if active.
7   Insert mode; 1 returned if insert mode active.
```

The second situation is that we do not always wish to have keyboard input "echoed" on the screen. A key press inputs a unique value to a location in memory. We are able to read displayable characters only because the operating system writes (echos) them via stdout. But there are also times, with games and passwords for instance, when we prefer not to see the input echoed. Sometimes we want the keyed input to initiate an immediate action of some kind, or we may wish the input to be saved in memory for later interaction with other data.

Standard Library Functions

There are several functions in the standard library for reading input from the keyboard. Which we make use of depends on our need, which will vary as the program advances. Definitions follow:

Echoed character input:

*** int getchar(void);**

#include <stdio.h> /* declares getchar function */

Reads the next character in the input stream associated with stdin. Chars are echoed as entered and saved in a buffer until the ENTER key is pressed.

Usage: int ch; ch = getchar();

Listing 1 is an example of using getchar() in a program. The keyed input is saved in an array and printed back. There is no need for the array, as will be seen in the example for putchar(chr). An array is needed only if we intend to use the input later in the program.

```
/*
** GET_CHAR.C
** A program illustrating buffered keyboard input.
*/
#include <stdio.h>
#define MAX 81
#define CLRSCRN "\033[2J"  /* ANSI screen clear */
main()
{
int i = 0;
char chr[MAX];
 puts(CLRSCRN);
 printf("Key in less than 80 characters and press ENTER.\n");
  while((chr[i] = getchar()) != '\n' && i < MAX)  i++;
    chr[i] = '\0';  i = 0;
     while(chr[i] != '\0')       {
      printf("%c",chr[i]); i++;  }
       exit(0);
}
```

Listing 1.

```
/*
** GET_CHE.C
** A program illustrating unbuffered keyboard input with
** echo.
*/
#include <stdio.h>
#include <conio.h>
#define MAX 81
#define CLRSCRN "\033[2J"  /* ANSI screen clear */
main()
{
 char chr;   /* Keyboard input char */
  puts(CLRSCRN);
   printf("Key in less than 80 characters.\n");
   printf("Press ENTER when done.\n");
    do  {
      (chr = getche());
        }
  while(chr != '\r'); /* do not use '\n' */
    exit(0);
}
```

Listing 2.

```
/*
** GET_CH.C
** A program illustrating unbuffered keyboard input without
** echo.
*/
#include <stdio.h>
#include <conio.h>
#define CLRSCRN "\033[2J"  /* ANSI screen clear */
main()
{
char chr;   /* Keyboard input char */
 puts(CLRSCRN);
  printf("Key in any number of characters.\n");
  printf("The screen will remain blank.\n");
  printf("Press Q to exit.\n");
   do  {
     (chr = getch());
        }
         while(chr != 'Q');
      exit(0);
}
```

Listing 3.

*** int getc(stdin);**

#include <stdio.h> /* declares getc function */

Same as getchar except the source must be provided. For the keyboard this is stdin.

Usage: int ch; ch = getc(stdin);

Listing 1 will run with getc(stdin) replacing getchar().

*** int getche(void);**

#include <conio.h> /* declares getche function */

Reads a character directly from the keyboard with no buffering. The program receives the input as soon as the key is pressed; the ENTER key is not pressed.

Listing 2 is an example of using getche() in a program. Note the comment not to use '\n' in place of '\r' to terminate the do loop. Using '\n' will put the program into an endless input situation.

Non-echoed input:

*** int getch(void);**

#include <conio.h> /* declares getch function */

Reads a character directly from the keyboard with no buffering AND no echoing. The program receives the input as soon as the key is pressed; the ENTER key is not pressed. Same as getche() except for no echo.

Listing 3 is an example of using getch() in a program.

Keyboard test functions:

*** int kbhit(void);**

#include <conio.h> /* declares kbhit function */

Checks to see if a key has been pressed. Returns zero if no key is pressed; otherwise a non-zero return. This function does not wait for a key to be pressed.

Listing 4 is an example of a program using kbhit() in conjunction with getch() to identify which of the ten function keys has been pressed. On running the program the screen is cleared, then no further screen action is observed until a key is pressed. If the pressed key is not one of the function keys an error message is displayed and the program exits.

String input:

*** char *gets(string)**

char *s;

#include <stdio.h> /* declares gets function */

This function is useful for interactive programs. Two requirements must be met when reading in a string. The first is a function to provide the reading. The second is there must be a location in memory in which to store the string. gets() reads the string until it encounters a newline, '\n'. The '\n' is not saved, it is replaced with the NULL character, '\0'. Listing 5 is an example of a program using gets() in an interactive keyboard application.

*** int scanf(char *format, ...);**

#include <stdio.h> /* declares scanf function */

Formatted keyboard input. Because of its complexity this function is described separately further on.

Screen Output

The stream concept simplifies program output to the screen. Four functions for displaying characters on the screen are putchar(chr), putc(chr,stdout), puts(string) and printf(). As with scanf(), printf() is complex compared to the others and is described later following scanf().

Character output:

*** int putchar(chr)**

#include <stdio.h>

char chr; /* The TURBO C Ref Guide declares chr as int, but programs compile and run with the char declaration. */

```
/*
** GET_FUNC.C
** A program illustrating non-ASCII keyboard input.
*/
#include <stdio.h>
#include <conio.h>
#define CLRSCRN "\033[2J"  /* ANSI screen clear */
#define FUNC1 "The key you have pressed is F1."
#define FUNC2 "The key you have pressed is F2."
#define FUNC3 "The key you have pressed is F3."
#define FUNC4 "The key you have pressed is F4."
#define FUNC5 "The key you have pressed is F5."
#define FUNC6 "The key you have pressed is F6."
#define FUNC7 "The key you have pressed is F7."
#define FUNC8 "The key you have pressed is F8."
#define FUNC9 "The key you have pressed is F9."
#define FUNC10 "The key you have pressed is F10."
#define DUMMY "Im not a function key, dummy!"
main()
{
int k_in_s, k_in_c;  /* k_in_s == 0, k_in_c == f_key code */
 puts(CLRSCRN);
 printf("Press any one of the 10 function keys.\n");
  do  {
   k_in_s = getch();
    if(k_in_s != 0) { puts(DUMMY); break; }
   k_in_c = getch();
        if(k_in_c == 59) puts(FUNC1);
   else if(k_in_c == 60) puts(FUNC2);
   else if(k_in_c == 61) puts(FUNC3);
   else if(k_in_c == 62) puts(FUNC4);
   else if(k_in_c == 63) puts(FUNC5);
   else if(k_in_c == 64) puts(FUNC6);
   else if(k_in_c == 65) puts(FUNC7);
   else if(k_in_c == 66) puts(FUNC8);
   else if(k_in_c == 67) puts(FUNC9);
   else if(k_in_c == 68) puts(FUNC10);
   else if(k_in_c >= 69 || k_in_c <= 58)
    puts(DUMMY);
       }
     while(!kbhit);
      exit(0);
}
```

Listing 4.

```
/*
** GET_S.C
** A program illustrating interactive keyboard string input.
*/
#include <stdio.h>
#define CLRSCRN "\033[2J"  /* ANSI screen clear */
main()
{
char name[81];  /* 80 chars plus closing NULL */
char *ptr, *gets();  /* declared as pointers  */
 puts(CLRSCRN);
  printf("Please type your full name and press ENTER.\n");
   ptr = gets(name);
    printf("\nSo you're %s? Welcome to my keyboard,
              %s.",name, ptr);
     exit(0);
}
```

Listing 5.

putchar is complementary to getchar(). This is illustrated in the simple program of Listing 6. When you run this program each key entry is echoed back immediately. On entering a 'Q' the do- while terminates, and the complete entry is repeated below the original. This because getchar() is buffered. To experiment try replacing getchar() with getch().

*** int putc(chr,stdin)**

#include <stdio.h>

char chr; /* As with putchar the TURBO C Ref Guide declares

chr as int, but programs compile and run with the char declaration. */. This function is essentially the same as putchar() with the added requirement to include the destination in the function call. This can be verified by replacing putchar(chr) in listing 6 with putc(chr,stdout).

String output:

*** int puts(string)**
char *string;
#include <stdio.h>

All the examples have made use of puts() for clearing the screen. It is simpler and easier to use than printf(), and also faster, but quite limited in what can be done. Each puts(string) will begin on a new line; this because when puts() detects a NULL char ('\0') it replaces it with the newline character, '\n'.

Listing 7 is a program applying puts() with an assortment of source lines. Observe the NULL character in the character array str4[]. Never write a character array for puts() without the closing NULL as it will not know when to stop without it.

Formatted Input and Output With SCANF and PRINTF

I am combining the discussion of these two functions because they uniquely complement each other. Our discussion of getchar(), getc(), getche(), getch() and gets() covered most of the keyboard situations that arise, but not all. Similarly for putchar(), putc(), putch() and puts(). The functions scanf() and printf() differ from these in providing for type specification and line formatting.

scanf

scanf() is derived from the general function fscanf() defined as:

int fscanf(FILE *stream, char *format[,argument,...]); to yield the keyboard input function:

```
int scanf(char *format[,argument,...]);
```

While fscanf() accepts its input from any stream pointed to by "stream" scanf() accepts its input only from stdin. The reference guide for TURBO C lists seven variations of scanf() which is far beyond the scope of this article. This writing applies to scanf(); fscanf() will be discussed in part 2. The format specifications direct the scanf() function to read and convert characters from the input field into specific types of values and store them in the locations given by the address arguments. The format specifications have the following forms:

```
% [ * ] [width] [F|N] [h|l] type_character
```

The format specification begins with the percent symbol (%). Not every form in the above is required in the format. If used, they are provided in the following sequence:

[*] an optional assignment-suppression character. Suppresses assignment of the next input field.
[width] an optional width specifier. Sets the maximum number of characters to be read.
[F|N] an optional pointer size-specifier. Overrides the default size of the address argument. N = near pointer, F = far pointer.

```
/*
** PUT_CHAR.C
** A program illustrating character output to the screen.
*/
#include <stdio.h>
#define CLRSCRN "\033[2J"
main()
{
 char chr;
  puts(CLRSCRN);
   do  {
   chr = getchar();  putchar(chr);
        }
    while(chr != 'Q');
     exit(0);
}
```

Listing 6.

```
/*
** PUT_S.C
** An example using the puts(string) function to display
** strings on the screen.
*/
#include <stdio.h>
#define CLRSCRN "\033[2J"
#define STR1 "You are viewing this string courtesy of puts."
main()
{
  static char str2[] = "I'm all strung out in an array.";
  char *str3 = "Now I'm the object of a pointer, tee hee!";
  static char str4[] = { 'B','y','e',' ','n','o','w','.','\0' };
   puts(CLRSCRN);
    puts(STR1);
    puts(str2);
    puts(str3);
    puts(str4);
     exit(0);
}
```

Listing 7.

```
/*
** GETRFUNC.C
** Reading a function key from the register values.
*/
#include <stdio.h>
#include <dos.h>
#define SCRNCLR "\033[2J"
#define INSTRCT "Press the F1 key."
#define WRONG   "That don't look like F1 from in here!"
#define RIGHT   "How about that, right on!"
#define SFKEY   0x16  /* interrupt 16H, keyboard I/O */
int scn, asc;
int rd_nonasky()
{
  union REGS regs;
    regs.h.ah = 0;
    regs.h.al = 0;
      int86(SFKEY, &regs,&regs);
 /* transfer register values to memory */
    asc = regs.h.ah;  /* ASCII code */
    scn = regs.h.al;  /* scan code  */
}
main()
{
   puts(SCRNCLR);
    puts(INSTRCT);
     rd_nonasky(scn,asc);
      if(asc != 59) puts(WRONG);
       else puts(RIGHT);
        exit(0);
}
```

Listing 8.

[h|l] an optional argument-type modifier. Overrides the default type of the address argument.

The type_character.

[F|N] and [h|l] are beyond the scope of this article.

The definition of input field applies to any one of the following:

1. All characters, up to but not including, the next whitespace character.
2. All characters up to the first one that cannot be converted in accordance with the existing format specification (an 8 or 9 under an octal format, for example).
3. Any number of characters within the specified field width.

Whenever input is to be used within our program it must be preserved in some manner. In Listing 1 we created an array to enable further use of getchar(). With scanf() we write the input directly to an address in memory. We specify the type_character, i.e., int, char, float, ... , of what is to be inputted to ensure the necessary memory is provided.

We frequently make use of scanf() in an interactive mode. That is, we may display a request such as :

```
puts(''Please enter your name:'');
scanf(''%15c'',&my_name);
```

When run, the screen displays:

```
Please enter your name:
```

and waits for our entry. If our name entry exceeds 15 characters it will be truncated on the screen.

In this call to scanf() %c is the type specification, 15 the maximum number of characters, and &my_name the storage location in memory. The most commonly used type specifiers are defined in Table 2.

Listing 9 illustrates usage of scanf(). If you experience a problem with your program skipping over the age query to print QUERY 3 without waiting for your Y or N input try replacing "c" with "s" in:

```
scanf(''%c'',&qry);
```

or, as another approach, delete the three lines relating to the name query and re-compile the program. My TURBO C version 1.5 compiles correctly with "%c" but runs correctly only if I replace "%c" with "%s." It will run correctly also with all reference to name removed.

I have found scanf() to have a variety of quirks -- it is not the easiest of functions to use. With Mix's POWER C, for instance, I had to declare the float variable as double, not float, and specify long float ("%lf") in the scanf() statement instead of simply "%f" in order to get the program to compile without error.

printf

Listing 9 provides three typical examples of applying printf() in our programs. In general, the specification list for scanf() applies equally to printf(). There are, however, additional features of printf() that can benefit our programs. Table 2 includes the specification and format coding for printf().

As with scanf(), printf() is derived from a general function:

```
int fprintf(FILE *stream,char *format[,argument,...]);
```

fprintf() places its argument on the named stream whereas printf() directs its output to stdout. There are six variations of printf() provided with TURBO C, the other five being cprintf(), sprintf(), vprintf(), vfprintf() and vsprintf(). Only printf() is discussed in this writing. cprintf(), like printf(), directs its output to the console, but does not translate line-feed characters into CR/LF combinations.

A floating point format specification not defined for scanf() is %g. With this specification the signed value will print in either ex-

```
/*
** SCANF_IN.C
** A program illustrating the use of scanf.
*/
#include <stdio.h>
#define CLRSCRN "\033[2J"
#define LF "\n"
#define QUERY1 "Hi! What name do you answer to?"
#define QUERY2 "Mind my asking your age? <Y/N>"
#define REPLY1 "Don't be a meanie. Tell me."
#define QUERY3 "C'mon, what is it?"
#define QUERY4 "How many dollars do you make in a month?"
main()
{
char name[], qry[];
int *age;
float pay;
  puts(CLRSCRN);
  puts(QUERY1);
  scanf("%s",&name);
  printf("%s? That's a great name!\n",name);
   puts(QUERY2);
   scanf("%c",&qry); /* may require change to %s. */
   if(*qry == 'Y'|| *qry == 'y') puts(REPLY1);
   puts(QUERY3);
    scanf("%i",&age);
    printf("Only %i? Just a babe in the woods!\n",age);
     puts(QUERY4);
     scanf("%6e",&pay);
     printf("Only %f? Wow! How about a new hard disk?\n",pay);
      exit(0);
}
```

Listing 9.

```
/*
** PRINTFX.C
** A program illustrating the use of printf.
*/
#include <stdio.h>
#define CLRSCRN "\033[2J"
#define HEX_AGE "Ah... That's better!\n"
main()
{
  char *FULL_NAME = "Thigmom O. Tourniquet";
  int my_age = 49;
  int my_son_age = 22;
  puts(CLRSCRN);
   printf("Hi! I am %s.\n",FULL_NAME);
   printf("My age in octal is %o.\n",my_age);
   printf("My age in hexadecimal is %x.\n",my_age);
   printf(HEX_AGE);
   printf("I was %d years old when my son was born.\n",my_age-my_son_age);
    exit(0);
}
```

Listing 10.

ponential (e or E) form, depending on the given value and its precision. An advantage is that the decimal point and trailing zeroes are printed only if necessary. To see this, try compiling and running Listing 9 with %g replacing %f for printing the monthly salary. That is:

```
scanf(''%6e'',&pay);
printf(''Only %g? Wow! How about a new hard disk?\n'',pay);
```

The *format term in the function prototype allows us to set conditions on the print field. That is, the unformatted %f specification in Listing 9 displays a half dozen trailing zeroes following the monthly salary. Replacing %f with %g eliminates the decimal and the zeroes. There is a better way: format %f. Revise this statement again by replacing "%f" with "%4.2f". Recompile and run, and on the salary query include a decimal point with more than two trailing digits. Bear in mind that scanf() will only accept up to 6 characters. The 4.2 sets a field width of four digits preceding the decimal point and 2 following it. So now:

```
scanf(''%6e'',&pay);
printf(''Only %4.2f? Wow! How about a new hard
disk?\n'',pay);
```

A point of interest about the field width specified is that scanf() will accept six numbers while a width of only four is provided in printf(). So, compile and run and make a variety of entries to see what happens. The field width with printf is flexible, when the number of entries exceeds the specified width, it adapts. So the true limit here is set by scanf().

Another formatting character that is useful to us is the minus sign (-). Normally when a quantity is printed from a specification, formatted or unformatted, the first character is all the way to the left. Now suppose we set a field width of 10 in our printf() statement. We only have six characters, so what happens with regard to the four blanks? Printf() starts the display with four leading blanks, that's what. Preceding the format with the minus sign instructs printf() to begin the display with the first character, eliminating the leading blanks. Try it with:

```
printf(''Only %-10.2f? Wow! How about a new hard
disk?\n'',pay);
```

Actually, there is more we can do with printf() that might be imagined. Like conversions between number bases and in-line arithmetic. Examples of these are provided in Listing 10.

Summary

At this time we have covered the most useful of the keyboard and video I/O library functions. Those that have not been discussed are narrower in their application and may not be available for all compilers. Part 2 will continue with disk file input/output and outputting control codes to our printer. ●

SCANF() FORMAT SPECIFICATIONS

SPEC	FUNCTION
c	character input. A field width W may be included; that is, %5c specifies an array of five characters.
s	a string input is read and stored in an array. Leading whitespace is not read. The string is input until the next whitespace; a space or newline terminates the input field. A NULL terminator is appended as the final array element.
i	a decimal, octal or hexadecimal integer.
l	a long decimal, octal or hexadecimal integer.
o	an octal integer.
d	a decimal integer.
u	unsigned decimal integer.
x	a hexadecimal integer.
e	floating point input.
f	floating point input.

PRINTF() FORMAT SPECIFICATIONS

SPEC	FUNCTION
c	variable to be printed is a character.
s	variable to be printed is a string.
i	variable to be printed is an integer.
l	variable to be printed is a long integer.
o	variable to be printed is an octal integer.
x	variable to be printed is a hexadecimal integer.
e	variable to be printed is floating point.
f	variable to be printed is floating point.
g	variable to be printed is floating point.
-	use with a field width definition to begin printing with the first non-blank character.

Table 2. Format specification listing for scanf() and printf().

The Z-System Corner

by Jay Sage

For some time I have been planning to discuss issues connected with setting up a remote access system (RAS), such as a Z-Node. There is still a great need for new nodes, and I think there are quite a few people toying with the idea of setting one up; they just aren't sure they know how to do it.

In fact, I am going to take up only a very small part of this question here -- and not even the part that would really help someone implement a system. After thinking about it, I realized that I am not the best person to talk about the procedures. I may not even be a good person. In fact, I'm not even sure that I know any longer how to do it!

To solve this problem, I am going to try to take the approach I have been taking more and more recently -- recruit someone else to write a TCJ article. Specifically, I hope to get the sysop(s) of one or more of the newer nodes to document the procedures they went through. After all, they are the real experts on the subject. My discussion here will instead cover only theoretical issues. I hope that even those who have no interest in setting up their own RAS will find such a discussion interesting.

As usual, before I get to the main technical topic, I have a number of other matters to cover first. Here is my list for this issue: Z-Node update, Z-Helpers, GEnie, and BDS C.

Z-Node Update

It has been three issues since the last update on the Z-Node roster, and there have been quite a few changes. The complete list is reproduced in Listing 1. Many inactive nodes have been dropped, and four new nodes have been added.

Chris McEwen's Socrates board in New Jersey is now Z-Node #32. The system is running under NZCOM on a Xerox 16/8 computer. Besides offering general support to the 8-bit community, Chris is a special resource for owners of Xerox computers. Unfortunately, the recent changes in the Newark outdial of PC-Pursuit have cut ZN32 off from a lot of its regular callers. As a result, Chris has switched to an alternative low-cost data transfer service called StarLink.

Briefly, StarLink provides much wider geographical access and full support for 2400 bps connections without the packet-switching delays encountered on PCP. It is more expensive than PCP at higher monthly usage rates but cheaper at lower usage levels. In the future we want to include in the Z-Node roster the StarLink addresses as well as PCP outdials. I'd appreciate it if anyone with such information would could get it to me using any of the methods indicated in the sidebar to my column.

The NYOUG (New York Osborne User Group) Fog #15 node (sysop Livingston Hinckley) has become a Z-Node as well, with the same number 15. The system is running on an Osborne Executive (CP/M-Plus) with the Z3PLUS version of Z-System. It is, thus, the first Z-Node -- though, I believe, not the first RAS -- to run under Z3PLUS. The board supports data rates up to 9600 bps using a USR Courier HST modem.

Dave Trainor in Cincinnati, Ohio, has signed up as Z-Node #7. His NZCOM-based system has many of the Z-System tools, such as VLU and VFILER, available for use on-line.

Greg Miner established a new frontier for Z-System as he became Z-Node #11 in Port Williams, Nova Scotia. Greg is fairly new to the Z-System but has already made a significant contribution as a programmer. He realized that on properly aliased Z-Systems, a "DIR *.COM" command does not really indicate to the user the commands that are available. So, he wrote ADIR (Alias DIRectory). It examines the ALIAS.CMD file, figures out the names of all the ARUNZ aliases (allowing for the multiply named scripts), and displays a sorted listing of them. A very fine program!

Finally, Ludo VanHemelryck, with assistance from Michael Broschat, will be setting up a new Z-Node in Seattle to replace Norm Gregory's system, which has gone over to MS-DOS. Recently, while on a business trip to Portland, Oregon, I had the pleasure of meeting Ludo and Michael (frankly, I was rather flattered that they were willing to drive all the way down from Seattle), and I think they will do a lot to boost Z-System interest in the Seattle area.

Z-Helpers

In the early days, getting ZCPR3 installed on a computer was a very arcane process, beyond the capability of most potential users. Echelon had the wisdom to compile a list of people all over the country (actually around the world) who were

---

Jay Sage has been an avid ZCPR proponent since the very first version appeared. He is best known as the author of the latest versions 3.3 and 3.4 of the ZCPR3 command processor and for his ARUNZ alias processor and ZFILER file maintenance shell.

When Echelon announced its plan to set up a network of remote access computer systems to support ZCPR3, Jay volunteered immediately. He has been running Z-Node #3 for more than five years and can be reached there electronically at 617-965-7259 (on PC-Pursuit). He can also be reached by voice at 617-965-3552 (between 11pm and midnight is a good time to find him at home) or by mail at 1435 Centre St., Newton, MA 02159. Finally, Jay recently became the Z-System sysop for the GEnie CP/M Roundtable and can be contacted as JAY.SAGE via GEnie mail.

In real life, Jay is a physicist at MIT, where he tries to invent devices and circuits that use analog *computation to solve problems in signal, image, and information processing.*

willing to help other users get through the process. These people were called Z-Helpers.

Today, thanks to NZCOM and Z3PLUS, getting the Z-System installed is very easy. It's even easy to get started using it, since it can be run just like CP/M. Taking full advantage of its capabilities, however, is another story. With a richness of function comparable to that of Unix, the Z-System can be daunting, but when one remains ignorant of its capabilities, one misses out on a lot of its utility and fun.

In looking over the Z-Helper list that is posted on the Z-Nodes and included with the NZCOM and Z3PLUS packages, I realized that this list has not been updated for many, many years, and most of its information is obsolete. It's high time that the list be rebuilt!

My plan is to discard the current list and start over, keeping only the few people I know to be active still. I would like to expand that list greatly. If you feel that you could be of some assistance to new Z-System users, please send me a post card or short note with the following information: name and address, voice phone number and hours, EMAIL addresses, if any (BBSs, GEnie, Compuserve, ARPAnet, BITNET, etc.), special areas of expertise, if any (specific computers, Z3PLUS). Remember, you do not have to be a Z-System know-it-all—none of us is. Willingness and eagerness to help are the most important qualifications of a Z-Helper, and by working with others, you will end up learning a lot yourself.

Listing 1. Z-Node List

```
                          Z-Node List #53
                  Sorted by State/Area Code/Exchange

Revised Z-Node list as of May 30, 1989.  An "R" in the left column
indicates a node that has registered with Z Systems Associates.  Report any
changes or corrections to Z-Node Central (#1) or to Jay Sage at Z-Node #3 in
Boston (or by mail to 1435 Centre St., Newton Centre, MA 02159-2469).

  NODE      SYSOP          CITY          STATE  ZIP    RAS Phone      PCP      Verified

Z-Node Central
--------------

R   1 Richard Jacobson  Chicago          IL   60606  312/649-1730  ILCHI/24  05/20/89
R   1 Richard Jacobson  Chicago          IL   60606  312/664-17330 ILCHI/24  05/20/89

Satellite Z-Nodes:
------------------

R   2 Al Hawley         Los Angeles      CA   90056  213/670-9465  CALAN/24  05/20/89
R   9 Roger Warren      San Diego        CA   92109  619/270-3148  CASDI/24  02/01/89
R  66 Dave Vanhorn      Costa Mesa       CA   92696  714/546-5407  CASAN/12  10/30/88
R  81 Robert Cooper     Lancaster        CA   93535  805/949-6404            12/29/88
R  36 Richard Mead      Pasadena         CA   91105  818/799-1632            11/01/88

R  17 Bill Biersdorf    Tampa            FL   33618  813-961-5747  FLTAM/24   (down)
      (node 17 expected to be up beginning of summer)

R   3 Jay Sage          Newton Centre    MA   02159  617/965-7259  MABOS/24  05/20/89

R  32 Chris McEwen      Plainfield       NJ   07080  201-754-9067  NJNEW/24  05/20/89

R  15 Liv Hinckley      Manhattan        NY   10129  212-489-7370  NYNYO/24  05/20/89

R   7 Dave Trainor      Cincinnati       OH   45236  513-791-0401            05/20/89

R  33 Jim Sands         Enid             OK   73703  405/237-9282            11/01/88
R  58 Kent R. Mason     Oklahoma City    OK   73107  405/943-8638

R   4 Ken Jones         Salem            OR   97305  503/370-7655            09/15/88
   60 Bob Peddicord     Selma            OR   97538  503/597-2852            11/01/88
R   8 Ben Grey          Portland         OR   97229  503/644-4621  ORPOR/12  05/20/89

R   6 Robert Dean       Drexel Hill      PA   19026  215-623-4040  PAPHI/24  05/20/89
R  38 Robert Paddock    Franklin         PA   16323  814/437-5647            11/01/88

R  77 Pat Price         Austin           TX   78745  512/444-8691            10/31/88
R  45 Robert K. Reid    Houston          TX   77088  713/937-8886  TXHOU/24  05/20/89

   10 Ludo VanHemelryck Seattle          WA         206/481-1371  WASEA/24
R  78 Gar K. Nelson     Olympia          WA   98502  206/943-4842            09/10/88

R  65 Barron McIntire   Cheyenne         WY   82007  307/638-1917            12/12/88

R   5 Christian Poirier Montreal Quebec   H1G 5G5 CANADA  514/324-9031      12/10/88
R  40 Terry Smythe  Winnipeg    Manitoba  R3N 0T2 CANADA  204/832-4593      11/01/88

R  62 Lindsay Allen  Perth, Western AUSTRALIA 6153        61-9-450-0200     12/21/88
   50 Mark Little    Alice Springs, N.T. AUSTRALIA 5750   61-089-528-852
```

GEnie CP/M Roundtable

Some of you may have noticed the change in the sidebar to my article listing a GEnie mailbox for me (JAY.SAGE). GEnie has been making a determined effort to provide support to the CP/M community. The indefatigable Keith Petersen is the main force behind it, and he recruited me several months ago to join the staff as a sysop specializing in the ZCPR3 and Z-System areas.

A system like GEnie offers a significant advantage over individual Z-Nodes: universal connection. With the Z-Nodes, if you want to leave me a message, you have to call one of the Z-Nodes that I call into regularly, and then you have to call around again to see if and where I might have left a response. Systems like GEnie and Compuserve offer central communication points readily accessible from most places in the US and Canada (and with worldwide access developing rapidly—Japan is on-line already).

One of the activities on GEnie is called a real-time conference (RTC), in which users can communicate interactively. These conferences are held on many subjects. The CP/M RTC takes place every Wednesday evening at 10 p.m. eastern time, with the first Wednesday session of each month generally led by me and earmarked for Z-System discussion. So, if you have questions or suggestions that you would like to discuss with me and other Z-System users, please consider joining us on the GEnie RTC. If you don't already belong to GEnie, check BBS systems near you for

information on a special GEnie signup offering that gives you $20 of free connect time as a new registrant.

BDS C Update

A couple of issues back I announced that a special Z-System version of BDS C would soon be available, and a number of people have been asking me about its status. The standard CP/M version 1.60 of BDS C has been available about a year already, and a working Z-System version is now being sold. We will probably eventually make a few more changes (such as adding support for type-3 program generation). In that case, everyone who ordered the earlier version will be offered an update at a price not to exceed the cost of media and shipping.

Some people have criticized the $90 price, comparing it to Turbo Pascal, which sells for only $60. What they don't realize is how much more one gets with BDS C for the extra $30. Listing 2 shows the contents of the four DSDD diskettes in the release, comprising well over 100 files and more than a megabyte of material!

First there are the core COM files you would expect: the compiler (CC.COM), the code generator (CC2.COM), the linker (CLINK.COM), and a librarian (CLIB.COM). CC, CC2, and CLINK come in both standard CP/M and Z-System versions, with separate run-time packages.

Then there is the stuff that almost no one gives you—the source code. True, you don't get the source for the core items, but you do get source code for everything else. Assembly-language source is included for the run-time package, and C source is provided for the collection of standard libraries. Some of the items are provided only in C source code; you have to compile them to produce COM files configured for your system and tastes. This includes two assemblers—one that uses Intel mnemonics (CASM) and one that uses Zilog mnemonics (ZCASM)—and a symbolic debugger (CDB). And while you don't get the source for CLINK, you do get the source for another linker, L2, that has even more capability (such as handling code that won't all fit in memory at one time).

RED, an editor with special hooks into the C error listing, is also provided, again in source form. You also get a number of sample programs, ranging from simple ones like CP (copy files) to an implementation of the MODEM7 file-transfer protocol (CMODEM.C). [P.S. If you absolutely and irresistibly crave the assembly-language source code to the BDS C core components (for personal use, not resale, of course), they can be purchased as an extra option for $200.]

Support is another important issue. Who supports Turbo Pascal? The same people who when asked about Turbo Modula 2 vigorously deny that they ever offered such a product at all! [Alpha Systems, unfortunately, acquired only the right to sell Turbo Pascal; Borland did not give them the source code and does not allow them to maintain it.] With BDS C, the situation is quite the opposite. On page 1 of the BDS C manual, at the top of the page, author Leor Zolman's personal phone number is listed, and he not only invites you to call him, day or evening, he actually is happy when you do!

In short, BDS C is a remarkable product from a remarkable programmer.

Listing 2. Files that comprise the four DSDD diskettes in the Z version of BDS C (directories captured using the BGii SCREEN command).

```
XD III  Version 1.2    Standard BDS C, Version 1.60
Filename.Typ Size K RS  Filename.Typ Size K RS  Filename.Typ Size K RS
-------- --- ------ --  -------- --- ------ --  -------- --- ------ --
CCC      .ASM     34     NOBOOT   .C        2     DEFF2B   .CSM     24
BUILD    .C        6     RM       .C        2     DEFF2C   .CSM      8
CASM     .C       24     STDLIB1  .C        8     WILDEXP  .CZ       6
CCONFIG  .C       12     STDLIB2  .C        8     CDBUPDAT.DOC       6
CCONFIG2.C         8     STDLIB3  .C        6     FILES    .DOC      4
CDB      .C        6     TAIL     .C        2     CCONFIG  .H        2
CHARIO   .C        4     UCASE    .C        2     CDB      .H        4
CLOAD    .C        4     C        .CCC      2     CDB1     .H        2
CMODEM   .C       12     CC       .COM     16     CMODEM   .H        2
CMODEM2  .C        8     CC2      .COM     18     DIO      .H        2
CP       .C        6     CDB      .COM     16     HARDWARE.H         4
DATE     .C        4     CLIB     .COM      6     STDIO    .H        2
DI       .C        4     CLINK    .COM      6     BDS      .LIB      6
DIO      .C       10     DEFF     .CRL     12     READ     .ME       4
L2       .C       26     DEFF2    .CRL      6     C        .SUB      2
LPR      .C        4     DEFF2A   .CSM     20     CASM     .SUB      2
     F  1:  --    48 Files Using   384K (   18K Left)

XD III  Version 1.2    Standard BDS C, Version 1.60
Filename.Typ Size K RS  Filename.Typ Size K RS  Filename.Typ Size K RS
-------- --- ------ --  -------- --- ------ --  -------- --- ------ --
ATBREAK  .C        4     RED3     .C       26     BCD2     .CSM     24
BMATH    .C       22     RED4     .C       20     DASM     .CSM      2
BREAK    .C        4     RED5     .C        4     BUGS     .DOC      2
CDB2     .C        2     RED6     .C        2     CRCK     .DOC      2
CDBCONFG.C         6     RED7     .C        4     BDSCIO   .H        4
COMMAND  .C        6     RED8     .C       14     CDB2     .H        4
DEMO1    .C        2     RED9     .C        2     MCONFIG  .H        2
LMATH    .C        2     TARGET   .C        4     RED      .H        6
LONG     .C        4     TSTINV   .C        2     RED1     .H        2
PARSE    .C       12     UTIL     .C        2     REDBUF   .H        4
PRINT    .C       10     CRCK     .COM      2     RED-READ.ME        4
RED10    .C       14     RCONFIG  .COM     24     CDB2     .OVL     20
RED11    .C       18     CRCKLIST.CRC       4     CRED     .SUB      2
RED12    .C       16     BCD      .CRL     16     L2RED    .SUB      2
RED13    .C        6     DASM     .CRL      2     LRED     .SUB      2
RED14    .C        6     DEFF15   .CRL     10     NULL     .SYM      0
RED2     .C       14     BCD1     .CSM      8
     F  2:  --    50 Files Using   376K (   18K Left)

XD III  Version 1.2    Special Z-System Files
Filename.Typ Size K RS  Filename.Typ Size K RS  Filename.Typ Size K RS
-------- --- ------ --  -------- --- ------ --  -------- --- ------ --
-BDSZ    .         0     CC2      .COM     18     DEFF2A   .CSM     20
CCC      .ASM     50     CCONFIG  .COM     20     DEFF2B   .CSM     24
CP       .C        6     CLINK    .COM      6     ZUTIL    .CSM      2
DI       .C        4     TAIL     .COM      6     STDIO    .H        2
ECH      .C        2     DEFF     .CRL     12     BDS      .LIB      8
TAIL     .C        2     DEFF2    .CRL      6     READ     .ME      18
WILDEXP  .C        8     DEFF3    .CRL      4     CASM     .SUB      2
C        .CCC      4     TAIL     .CRL      2     NDEFF2   .SUB      2
CC       .COM     16
     F  1:  --    25 Files Using   244K (  420K Left)

XD III  Version 1.2    Zilog-Mnemonic Version of Assembler
Filename.Typ Size K RS  Filename.Typ Size K RS  Filename.Typ Size K RS
-------- --- ------ --  -------- --- ------ --  -------- --- ------ --
-ZCASM   .         0     ZCASM    .DOC      8     READ     .ME       2
ZCASM    .C       24     --READ--.ME        2     ZCASM    .SUB      2
ZCASM    .COM     18
     F  2:  --     7 Files Using    56K (  420K Left)
```

Remote Access Systems

I had hoped by now to have gone through the process of completely revamping the RAS software on my Z-Node. It has been many years since I designed that system, and I really do not remember all the details of what I went through. Being the sort I am, I made a great number of custom modifications to all the programs, and, as a result, I have been stuck using old versions of everything. I use a derivative of BYE503 when the latest version is BYE520; I run a modified KMD09 when KMD has not only advanced to a much higher version number but has really been superseded by Robert Kramer's excellent ZMD. Admittedly, I already incorporated a number of the improvements that these versions offer; nevertheless, my node is quite outmoded.

With the TCJ column as an excuse to look into all these new developments, I thought I would be able to modernize Z-Node #3 and be able to tell you how to go about creating a new Z-Node in the easiest possible way. Alas, as usual, I have been too busy with other things. As I said earlier, I hope to get one or more of the new Z-Node sysops to contribute articles to TCJ on this subject.

Since I can't give a prescription for creating a standard remote access system (RAS) using the current software, I will instead discuss some of the basic concepts behind a remote access system and the ways in which Z-System facilities can be used to advantage. The standard RAS software is designed to work under standard CP/M and is far less efficient than it could be.

I/O Redirection

Once you have a secure Z-System running, as we described last time, the next step is to make it possible for the system to be operated via the modem. The standard software that does this is called BYE, and it traces its roots all the way back to the work of Keith Petersen and others from the days of Ward Christensen's first remote CP/M (or RCPM) system in the late 1970s.

A great deal of development has occurred since that time, and BYE now provides a rich array of services. Its essential function, however, is to provide redirection of the console input/output functions. Program input is allowed to come not only from the local keyboard but also from the modem; program output is sent not only to the local screen but also to the modem. In this way, either the local operator or the person connected via the modem can operate the computer. With a secure Z-System, this alone would be enough for a rudimentary RAS.

BYE works by installing itself as an RSX, or resident system extension, generally just below the command processor. It patches the data in page 0 of memory (this is where programs find out how to request services from the operating system) and in the actual BIOS. These patches do two things. They protect BYE so that a warmboot will not result in its removal from memory, and they allow BYE to intercept software calls to the BIOS and DOS from any running programs. BYE can then substitute its own additional or special functions.

In a Z-System, the extended character I/O functions of BYE could properly be implemented using an IOP (Input/Output Package). The IOP is a generalization of the concept from early CP/M of the so-called IOBYTE, which was controlled by the STAT command and used to select from a fixed set of I/O devices (up to four possibilities in the case of the console). Richard Conn conceived of the IOP module as a way to handle just the kind of I/O operations needed for a RAS, and his book, "ZCPR3, The Manual," has some examples of this. Alternatively, the BIOS functions could be implemented directly in an NZCOM VBIOS (virtual BIOS), which could be loaded as needed.

Consider the case where the console is an external terminal connected to an RS232 serial port. The standard BIOS CONOUT (console output) function takes the character in register C and sends it to that serial port. For a remote system, the BIOS would send the character first to the terminal's serial port and then to the modem's serial port.

Similarly, the standard BIOS CONST (console status) function checks the terminal's serial port to see if a character has been received. If so, it returns with a nonzero value in register A; otherwise it returns a zero value. For a remote system, the CONST routine would check both serial ports and return a nonzero value if either one has a character ready. CONIN (console input) would poll the two serial ports alternately until it got a character from one or the other of them.

Conceptually, this is really all pretty straightforward. The biggest problem is that the code depends on the specific computer hardware, so no universal routines can be supplied. The BYE program has been cleverly designed, like an operating system, with the hardware-specific code separated from the hardware-independent code. As a result, customized versions of BYE can be assembled easily by putting source code inserts for one's specific hardware at designated places in the master file. There are two collections of inserts, one for many computer types and one for many types of real-time clocks (more about this later).

There are also some fine points about how the console redirection is handled. BYE, of course, knows the difference between the local console and the modem and can treat them differently where appropriate. For example, since modems commonly produce certain noise characters that rarely appear in intended input (such as the left curly brace), these can be filtered out (i.e., ignored). Also, some special functions can be assigned to local control codes. For example, pressing control-N at the local console will immediately end the callers session (used when the sysop doesn't like what a user is doing); on the other hand, control-U removes any connect time limit, and control-A allows special access by turning on the wheel byte (toggling it, actually).

The screen output can likewise be handled differently. Pressing control-W locally causes a local display of the current caller's name. Something I added to my version of BYE was a filter that prevents escape sequences from going to the local console. I allow (encourage) users to take advantage on-line of virtually all Z-System capabilities, including the full-screen utilities like ZFILER, ZMANAGER, and VLU. When I first started to do this, I found that the escape sequences going to the users' terminals sometimes did very bad things to my own terminal (the smartest terminals, like the Wyse and Televideo models, have the worst problems; they have sequences—that cannot be disabled—that lock out the keyboard!). I simply borrowed the code used in BYE for the incoming-character filter.

Modem Initialization and Call Detection

The first real complication we have to face is that the actions described above make sense only after the local modem is in communication with a remote modem. Consequently, BYE has traditionally been given the additional task of initializing the modem and monitoring it for an incoming call. A 'smart modem' (one that processes the Hayes "AT" commands and returns the Hayes result codes) is generally assumed, though BYE apparently will work with other modems to some extent.

The standard procedure today for answering calls (unless it has changed again since my BYE503) is not, as one might have expected, to set the modem to auto-answer mode. There are some subtle reasons why this is less desirable. Instead, BYE monitors the modem port for an

SAGE MICROSYSTEMS EAST

Selling & Supporting the Best in 8-Bit Software

- New Automatic, Dynamic, Universal Z-Systems
  - Z3PLUS: Z-System for CP/M-Plus computers ($69.95)
  - NZ-COM: Z-System for CP/M-2.2 computers ($69.95)
  - ZCPR34 Source Code: if you need to customize ($49.95)
- Plu*Perfect Systems
  - Backgrounder II: switch between two or three running tasks under CP/M-2.2 ($75)
  - ZDOS: state-of-the-art DOS with date stamping and much more ($75, $60 for ZRDOS owners)
  - DosDisk: Use DOS-format disks in CP/M machines, supports subdirectories, maintains date stamps ($30 - $45 depending on version)
- BDS C — Special Z-System Version ($90)
- SLR Systems (The Ultimate Assembly Language Tools)
  - Assembler Mnemonics: Zilog (Z80ASM, Z80ASM+), Hitachi (SLR180, SLR180+), Intel (SLRMAC, SLRMAC+)
  - Linkers: SLRNK, SLRNK+
  - TPA-Based: $49.95; Virtual-Memory: $195.00
- NightOwl Software MEX-Plus ($60)

Same-day shipping of most products with modem download and support available. Order by phone, mail, or modem. Shipping and handling $4 per order (USA). Check, VISA, or MasterCard. Specify exact disk format.

Sage Microsystems East
1435 Centre St., Newton Centre, MA 02159-2469
Voice: 617-965-3552 (9:00am - 11:30pm)
Modem: 617-965-7259 (password = DDT)(MABOS on PC-Pursuit)

output string from the modem indicating that it has detected a ring signal. This string will be "RING" if the modem is in verbose mode or "2" if it is in terse mode. BYE then sends the command "ATA" to the modem so that it will answer the call. Then BYE waits for another string that indicates the data rate of the connection; in terse mode these are "1" for 300 bps, "5" for 1200 bps, "10" for 2400 bps, and other values for higher speed or error-correcting modems. Finally, the serial port is synchronized to the modem's data rate.

If no connect indication is received within a prescribed time, BYE recycles the modem to make it ready for another call. Once a connection has been established, BYE generally displays a welcome message, and it may ask for a password. Then it loads an initial program, typically the bulletin board program.

The functions we just described do not really have to be handled in BYE at all, and TPA could be saved (remember, the BYE code remains resident in high memory at all times) if they were performed by a transient program. With a Z-System there is also no need for BYE itself to load a program file into memory. All it has to do is place a command line into the multiple command line buffer. This requires much less code and is faster and more flexible. Moreover, the full power of the command processor is available to locate and load the program code. Resident, type-3, and type-4 programs can be used. I modified my BYE to load the simple command line STARTRAS, which, as you probably guessed, is an alias (it can be an ARUNZ alias, in fact). This makes it very easy to make changes and experiment.

Carrier Monitoring

There is one further function of BYE that cannot easily be performed anywhere else. That is monitoring the carrier-detect line for a loss of connection. After all, a caller might hang up or become disconnected at any time. Some special clean-up functions must then be performed. The currently running program must be aborted, any system maintenance functions performed (such as updating the database of caller information), and the modem must be reset and readied for another call. Without going into any detail, I do want to point out that terminating a running program can be tricky, especially if file I/O is in progress.

Besides detecting when a user disconnects, either voluntarily or accidentally, BYE can also enforce a time limit on the caller. For many of its timing functions, BYE requires a real-time clock. As I mentioned earlier, there is a library of clock inserts that can be installed in the BYE source before it is assembled.

This completes the discussion for this issue. So far we have touched on the BIOS and modem-control functions of BYE; next time I plan to take up the DOS services that BYE provides. •

Real Computing

The National Semiconductor NS32032

by Richard Rodman

What, I have been asked, is "Real Computing"? Real computing is where performance counts, where work is done, where people think and machines work. Real computing is where the machine fills a purpose in a larger system, such as a company or a factory.

This time I'm going to talk about embedded computing, where programmed control is embedded within the structure of a larger machine. More often than not, the "user interface" to such a processor is very limited, such as a button or two, a couple of lights, a few TTL or analog I/O lines. The operation may be completely invisible to an outside observer.

Many *TCJ* readers work in the embedded computing environment, as shown by their interest in interfacing unusual devices. Those of you who do not should consider this: About 10 million PCs have been sold in ten years. More than that many embedded processors are installed *each year.* Furthermore, the applications software market fluctuates with the fortunes of the PC vendors; a big downturn occurs every couple of years. The embedded computing market has grown every year since the 4004 came out.

Embedded Systems

National Semiconductor has decided to reorientate their marketing plan to embedded systems, instead of to the more glamorous Unix, workstation or PC markets. Obviously, a lot more chips are sold in that market.

There are basically three sub-markets in the embedded computer market. From the low-end to the high-end, each sub-market is about one-tenth the size of the one below it.

The low-end sub-market is ruled by 4- and 8-bit microcontrollers, principally the 8051. These processors typically have some on-chip memory and peripherals, and an instruction set with "hard-wired" support for the on-chip I/O. While C compilers are available, no real programmer would use one. These chips have to be programmed in assembler to achieve any semblance of performance. Devices in this section are sold in tens of millions of units each year, and control simple devices such as microwave ovens, keyboards, . . . No bus is ever used in these devices, because this sub-market is extremely cost-sensitive: a few pennies difference might dictate the choice of a processor.

The mid-level sub-market is the broadest region of the market, and it is almost completely ruled by the Z-80. These processors are used in areas where greater performance or more memory is needed, such as in industrial process controls, data buffers, disk and printer controllers, fax machines, high-speed modems, multiplexers, printer buffers, plotters, and so forth. Programming is mostly in assembler, because C compilers for the Z-80 exact too high a performance penalty. Buses are sometimes used in these devices, which are 8-bit buses such as the G64 and STD buses. Because of high cost-sensitivity, 8 bits is the main bus size, and that isn't likely to change.

The high-end sub-market is used in high-performance, real-time device control or data manipulation systems. Processors in this market need high performance, fast interrupt response, and simple and fast access to, and manipulation of, large memory buffers. These actually have a broad range of application such as laser printers, X.25 multiplexers and packet switches, synchronous data link protocol converters or front-ends, laboratory data acquisition systems, signal processing, image processing, rasterizing, and so on. In this market, the ruling processor is the 68020, and the ruling bus is the VME bus. However, the number of units is only in the hundreds of thousands. These systems are programmed about half in assembler, half in C, depending on whether high performance or quick time-to-market is the overriding concern.

Intel has made a little inroads into the mid-level market with the 80186 and 80188, and has been struggling to enter the high-level market with the 80386, 80376 and 80960. However, their segmented architecture has been a significant limiting factor.

Another approach which has gained some attention is the use of Forth in the mid-level and high-end submarkets. Harris has targeted their RTX-2000 chip specifically to the high-end submarket.

National has targeted the low-end with their COP and HPC processors, the mid-range with the 32008 processor, and the high-end with the 32016, 32CG16 graphics processor, 32332, 32532, and the new 32GX32 processor.

Another factor in the embedded systems market is the availability and quality of good, inexpensive development tools. This means that chips and boards must be available in experimenter's quantities at low prices, and development software must be available for common computing platforms at low cost. There are a number of low-cost boards available for the NS32 chip sets, or you can "roll your own." As for software, the C compilers and assemblers available from *TCJ* are first rate.

The 32GX32 CPU

In April, National announced the 32GX32 CPU. Many people assumed that, because of the letters in the middle, this part was very similar to the 32CG16. This, however, is not the case. Embedded computing usually needs a "hot rod" CPU, where the power steering and electric windows have been removed, and that's what the 32GX32 is--simply put, it's a 32532 without an MMU. By removing the MMU, National dramatically reduced the number of transistors on the chip, making it smaller and therefore less expen-

sive to produce. It still has the on-chip data and instruction caches.

In the same month, Motorola announced the 68332, which is a 68020 CPU with some I/O and a peculiar timer circuit added. This device, and its timer, was specifically designed for one particular high-volume customer, so whether the part appeals to anyone else is probably immaterial. National has indicated that they are willing to undertake similar projects based on the 32GX32.

Although National has been very careful to never mention the 32532 and the 32GX32 in the same breath, very interestingly, the two parts are pin-compatible, so that it would be possible to upgrade a system based on the 32GX32 to a 32532 in the future.

Assembly Language

You'll notice that, in embedded systems, assembly language is used except in the very high-end. In the application software world, assembly language is disdained as being non-portable. It's true, assembly language is not portable (although there have been attempts made at "universal assembly language"). Tricks and special techniques used in specific processor instruction sets are also non-portable.

However, assembly language *skills* are in fact quite portable. Thus, the real programmer working in assembly language must develop techniques that he can reuse. Some of these might be a tool box of common algorithms, or a naming convention for global variables.

Assembly language is a good place to learn how to really make your software reliable. In C, it is possible to limit the scope of variables and functions by calling them "static" (Dennis Ritchie saved adding a new keyword by reusing an old one—a penny-wise, pound-foolish decision all C programmers suffer from), but almost nobody does it. The assembly language programmer learns that a global variable is like an open door—you never know who might come in there—so he wisely limits the scope of variables as much as possible, and combines functions into modules in such a way as to reduce the number of globals.

Further, an assembly language program is a lot bigger—in terms of lines of source—than a C program. To keep the size in bounds, the programmer learns to make subroutines which are general, taking parameters, so that they can be used by many different parts of the program. C programmers often take a "when in doubt, write another function" approach.

The main problem with the C language is that it is caught in the middle between a high-level language like Ada and low-level assembler. In some cases (for example, the freeness of "casting" pointers or performing pointer arithmetic), it follows the assembler philosophy; in other cases, such as arithmetic expressions, it follows high-level thinking. The result is a schizophrenic language that acts in bizarre ways. For example:

```
long longvar1, longvar2;
longvar1 = (( long ) func1() + longvar2 ) + 2;
```

Even though func1 is casted to long and added to a long and the result is a long, the compiler sees a single int-sized quantity (the 2) and truncates the intermediate result to an int, just before sign-extending it to store it in longvar1.

Assembly language makes no pretensions of any kind. It's explicit. It says what it means and means what it says, nothing more, nothing less.

"But assembly language is hard," some might say. While the Intel family of processors seems to have been fiendishly designed to make assembly language frustrating and confusing, the NS32 instruction set is simple and easy to pick up. Why, your first program might run the first time.

Next time

Next time I'll return to the NS32 architecture and discuss the trap mechanism, including the built-in single step mechanism. Plus, I'll cover some new information about Neil Koozer's 32HL, originally described in issue #30. •

Where to write or call:

Richard Rodman
8329 Ivy Glen Court
Manassas VA 22110
BBS: 703-330-9049

Technology Resources

K-OS ONE—Single user generic 68000 operating system for your 68000 hardware. It uses the MS-DOS disk format, and includes the operating system with source code (written in HTPL), an editor, assembler, and HTPL compiler. A sample BIOS code and a boot loader are included. This is not ready-to-run—you have to install the BIOS on your system, but the source code and language compiler are included **$50**

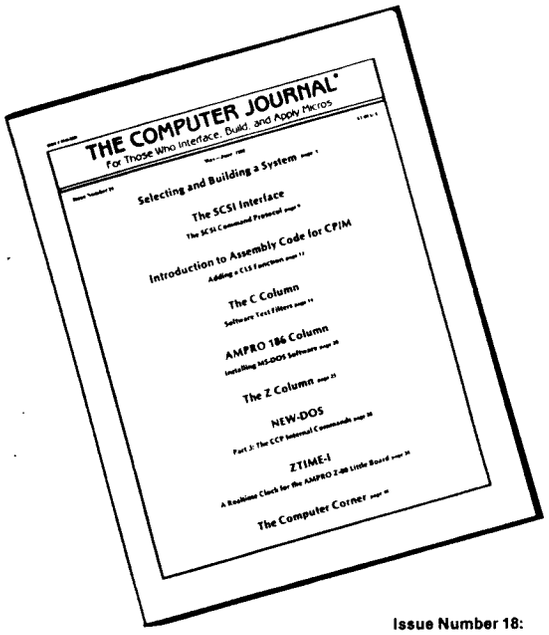
HT-Forth—A full featured, interactive Forth that works with the K-OS ONE operating system. It uses a full 32 bit stack and 32 bit arithmetic to take full advantage of the 68000. Programs are position independent and are limited in size only by the memory available. Source code compiles to inline macros, JSR, or BSR so there is no inner interpreter overhead. Standard ASCII files are used. Includes full screen editor and a Forth style 68000 assembler **$100**

68000Cross Assembler—Written entirely in 8086 assembly language, it is small and fast. All input and output is done with standard MS-DOS calls so it will run on any MS-DOS system, even those which are not totally PC compatible. All 68000 and 68010 instructions are supported. It has conditional assembly, the symbol table is in alphabetical order, and cross referencing is included. Include files are supported so it is easy to assemble big programs, but edit them in small pieces. An equate file can be produced for PROM based programming **$50**

ORDER FROM

The Computer Journal
190 Sullivan Crossroad
Columbia Falls, MT 59912
Phone (406) 257-9119

Visa and Mastercard accepted
Prices postpaid in the U.S. and Canada



THE COMPUTER JOURNAL

Back Issues

Issue Number 1:

- RS-232 Interface Part One
- Telecomputing with the Apple II
- Beginner's Column: Getting Started
- Build an ""Epram"

Issue Number 2:

- File Transfer Programs for CP/M
- RS-232 Interface Part Two
- Build Hardware Print Spooler: Part 1
- Review of Floppy Disk Formats
- Sending Morse Code with an Apple II
- Beginner's Column: Basic Concepts and Formulas

Issue Number 3:

- Add an 8087 Math Chip to Your Dual Processor Board
- Build an A/D Converter for Apple II
- Modems for Micros
- The CP/M Operating System
- Build Hardware Print Spooler: Part 2

Issue Number 4:

- Optronics, Part 1: Detecting, Generating, and Using Light in Electronics
- Multi-User: An Introduction
- Making the CP/M User Function More Useful
- Build Hardware Print Spooler: Part 3
- Beginner's Column: Power Supply Design

Issue Number 6:

- Build High Resolution S-100 Graphics Board: Part 1
- System Integration, Part 1: Selecting System Components
- Optronics, Part 3: Fiber Optics
- Controlling DC Motors
- Multi-User: Local Area Networks
- DC Motor Applications

Issue Number 16:

- Debugging 8087 Code
- Using the Apple Game Port
- BASE: Part Four
- Using the S-100 Bus and the 68008 CPU
- Interfacing Tips & Troubles: Build a ""Jellybean" Logic-to-RS232 Converter

Issue Number 18:

- Parallel Interface for Apple II Game Port
- The Hacker's MAC: A Letter from Lee Felsenstein
- S-100 Graphics Screen Dump
- The LS-100 Disk Simulator Kit
- BASE: Part Six
- Interfacing Tips & Troubles: Communicating with Telephone Tone Control, Part 1

Issue Number 19:

- Using The Extensibility of Forth
- Extended CBIOS
- A $500 Superbrain Computer
- BASE: Part Seven
- Interfacing Tips & Troubles: Communicating with Telephone Tone Control, Part 2
- Multitasking and Windows with CP/M: A Review of MTBASIC

Issue Number 20:

- Designing an 8035 SBC
- Using Apple Graphics from CP/M: Turbo Pascal Controls Apple Graphics
- Soldering and Other Strange Tales
- Build a S-100 Floppy Disk Controller: WD2797 Controller for CP/M 68K

Issue Number 21:

- Extending Turbo Pascal: Customize with Procedures and Functions
- Unsoldering: The Arcane Art
- Analog Data Acquisition and Control: Connecting Your Computer to the Real World
- Programming the 8035 SBC

Issue Number 22:

- NEW-DOS: Write Your Own Operating System
- Variability in the BDS C Standard Library
- The SCSI Interface: Introductory Column
- Using Turbo Pascal ISAM Files
- The AMPRO Little Board Column

Issue Number 23:

- C Column: Flow Control & Program Structure
- The Z Column: Getting Started with Directories & User Areas
- The SCSI Interface: Introduction to SCSI
- NEW-DOS: The Console Command Processor
- Editing The CP/M Operating System
- INDEXER: Turbo Pascal Program to Create Index
- The AMPRO Little Board Column

Issue Number 24:

- Selecting and Building a System
- The SCSI Interface: SCSI Command Protocol
- Introduction to Assembly Code for CP/M
- The C Column: Software Text Filters
- AMPRO 186 Column: Installing MS-DOS Software
- The Z Column
- NEW-DOS: The CCP Internal Commands
- ZTIME-1: A Realtime Clock for the AMPRO Z-80 Little Board

Issue Number 25:

- Repairing & Modifying Printed Circuits
- Z-Com vs Hacker Version of Z-System
- Exploring Single Linked Lists in C
- Adding Serial Port to Ampro L.B.
- Building a SCSI Adapter
- New-Dos: CCP Internal Commands
- Ampro '186 Networking with SuperDUO
- ZSIG Column

Issue Number 26:

- Bus Systems: Selecting a System Bus
- Using the SB180 Real Time Clock
- The SCSI Interface: Software for the SCSI Adapter
- Inside AMPRO Computers
- NEW-DOS: The CCP Commands Continued
- ZSIG Corner
- Affordable C Compilers
- Concurrent Multitasking: A Review of DoubleDOS

Issue Number 27:

- 68000 TinyGiant: Hawthorne's Low Cost 16-bit SBC and Operating System
- The Art of Source Code Generation: Disassembling Z-80 Software
- Feedback Control System Analysis: Using Root Locus Analysis and Feedback Loop Compensation
- The C Column: A Graphics Primitive Package
- The Hitachi HD64180: New Life for 8-bit Systems
- ZSIG Corner: Command Line Generators and Aliases
- A Tutor Program for Forth: Writing a Forth Tutor in Forth
- Disk Parameters: Modifying The CP/M Disk Parameter Block for Foreign Disk Formats

Issue Number 28:

- Starting your Own BBS
- Build an A/D Converter for the Ampro L.B.• HD64180: Setting the wait states & RAM refresh, using PRT & DMA
- Using SCSI for Real Time Control
- Open Letter to STD-Bus Manufacturers
- Patching Turbo Pascal
- Choosing a Language for Machine Control

Issue Number 29:

- Better Software Filter Design
- MDISK: Adding a 1 Meg RAM disk to Ampro L.B., part one.
- Using the Hitachi HD64180: Embedded processor design.
- 68000: Why use a new OS and the 68000?
- Detecting the 8087 Math Chip
- Floppy Disk Track Structure
- The ZCPR3 Corner

Issue Number 30:

- Double Density Floppy Controller
- ZCPR3 IOP for the Ampro L.B.
- 3200 Hacker's Language
- MDISK: 1 Meg RAM disk for Ampro LB, part 2
- Non-Preemptive Multitasking
- Software Timers for the 68000
- Lilliput Z-Node
- The ZCPR3 Corner
- The CP/M Corner

Issue Number 31:

- Using SCSI for Generalized I/O
- Communicating with Floppy Disks: Disk parameters and their variations.
- XBIOS: A replacement BIOS for the SB180.
- K-OS ONE and the SAGE: Demystifing Operating Systems.
- Remote: Designing a remote system program.
- The ZCPR3 Corner: ARUNZ documentation.

Issue Number 32:

- Language Development: Automatic generation of parsers for interactive systems.
- Designing Operating Systems: A ROM based O.S. for the Z81.
- Advanced CP/M: Boosting Performance.
- Systematic Elimination of MS-DOS Files: Part 1, Deleting root directories & an in-depth look at the FCB.
- WordStar 4.0 on Generic MS-DOS Systems: Patching for ASCII terminal based systems.
- K-OS ONE and the SAGE: Part 2, System layout and hardware configuration.
- The ZCPR3 Corner: NZCOM and ZCPR34.

Issue Number 33:

- Data File Conversion: Writing a filter to convert foreign file formats.
- Advanced CP/M: ZCPR3PLUS, and how to write self relocating Z80 code.
- DataBase: The first in a series on data bases and information processing.
- SCSI for the S-100 Bus: Another example of SCSI's versatility.
- A Mouse on any Hardware: Implementing the mouse on a Z80 system.
- Systematic Elimination of MS-DOS Files: Part 2—Subdirectories and extnded DOS services.
- ZCPR3 Corner: ARUNZ, Shells, and patching WordStar 4.0

Issue Number 34:

- Developing a File Encryption System: Scramble data with your customized encryption/password system.
- DataBase: A continuation of the database primer series.
- A Simple Multitasking Executive: Designing an embedded controller multitasking system.
- ZCPR3: Relocatable code, PRL files, ZCPR34, and Type 4 programs.
- New Microcontrollers Have Smarts: Chips with BASIC or Forth in ROM are easy to program.
- Advanced CP/M: Operating system extensions to BDOS and BIOS, RSXs for CP/M 2.2.
- Macintosh Data File Conversion in Turbo Pascal.

Issue Number 35:

- All This & Modula-2: A Pascal-like alternative with scope and parameter passing.
- A Short Course in Source Code Generation: Disassembling 8086 software to produce modifiable assem. souce code.
- Real Computing: The National Semiconductor NS32032 is an attractive alternative to the Intel and Motorola CPUs.
- S-100 Eprom Burner: a project for S-100 hardware hackers.
- Advanced CP/M: An up-to-date DOS, plus details on file structure and formats.
- REL-Style Assembly Language for CP/M and Z-System: Part 1-selecting your assembler, linker, and debugger.
- ZCPR3 Corner: How shells work, cracking code, and remaking WordStar 4.0.

Issue Number 36:

- Information Engineering: Introduction
- Modula-2: A list of reference books
- Temperature Measurement & Control: Agricultural computer application
- ZCPR3 Corner: Z-Nodes, Z-Plan, Amstrand computer, and ZFILE!
- Real Computing: NS32032 hardware for experimenter, CPU's in series, software options
- SPRINT: A review
- ZCPR3's Named Shell Variables
- REL-Style Assembly Language for CP/M & Z-Systems, part 2
- Advanced CP/M: Environmental programming

Issue Number 37:

- C Pointers, Arrays & Structures Made Easier: Part 1, Pointers
- ZCPR3 Corner: Z-Nodes, patching for NZCOM, ZFILER
- Information Engineering: Basic Concepts; fields, field definition, client worksheets
- Shells: Using ZCPR3 named shell variables to store date variables
- Resident Programs: A detailed look at TSRs & how they can lead to chaos
- Advanced CP/M: Raw and cooked console I/O
- Real Computing: NS320XX floating point, memory management, coprocessor boards, & the free operating system
- ZSDOS-Anatomy of an Operating System: Part 1

Issue Number 38:

- C Math: Handling Dollars and Cents With C.
- Advanced CP/M: Batch Processing and a New ZEX.
- C Pointers, Arrays & Structures Made Easier: Part 2, Arrays.
- The Z-System Corner: Shells and ZEX, new Z-Node Central, system security under Z-Systems.
- Information Engineering: The portable Information Age.
- Computer Aided Publishing: Introduction to publishing and Desk Top Publishing.
- Shells: ZEX and hard disk backups.
- Real Computing: The National Semiconductor NS320XX.
- ZSDOS--Anatomy of an Operating System, Part 2.

Issue Number 39:

- Programming for Performance: Assembly Language techniques.
- Computer Aided Publishing: The Hewlett Packard LaserJet.
- The Z-System Corner: System enhancements with NZCOM.
- Generating LaserJet Fonts: A review of Digi-Fonts.
- Advanced CP/M: Making old programs Z-System aware.
- C Pointers, Arrays & Structures Made Easier: Part 3: Structures.
- Shells: Using ARUNZ alias with ZCAL.
- Real Computing: The National Semiconductor NS320XX.
- The Computer Corner.

TCJ ORDER FORM

Subscriptions		U.S.	Canada	Surface Foreign	Total
6 issues per year ☐ New ☐ Renewal	1 year	$16.00	$22.00	$24.00	
	2 years	$28.00	$42.00		
Back Issues		$3.50 ea.	$3.50 ea.	$4.75 ea.	
Six or more #'s		$3.00 ea.	$3.00 ea	$4.25 ea.	
				Total Enclosed	

All funds must be in U.S. dollars on a U.S. bank.

☐ Check enclosed ☐ VISA ☐ MasterCard Card # _____

Expiration date_____ Signature _____

Name _____

Address _____

City_____ State_____ ZIP _____

THE COMPUTER JOURNAL

190 Sullivan Crossroad, Columbia Falls, MT 59912 Phone (406) 257-9119

#40

(Continued from page 4)

8.5 x 11, refers to the cut paper size. The logical area (also called the addressable area) which defines where the cursor can be positioned, and the printable area which defines where the printer can place a dot, are smaller. The manuals include a chart of the logical and printable areas. For 8.5 x 11 physical size, the logical area is 8 x 11, and the printable area is 8.16 x 10.66.

The LaserJet IID manual states that it performs pixel level clipping at the printable area boundaries so that any portion of a character or graphic which is outside of the printable area will be clipped. I am using a LaserJet II (not IID), and I lose the entire character if a portion of the character extends beyond the printable area boundary.

The cursor positioning commands place the cursor at the lower left hand corner of the character cell, and the character extends up and to the right of the cursor position. For raster graphics the cursor position is at either the upper left or upper right corner of the graphics cell. Keep the position of the character or graphics cell with relation to the cursor in mind before you place the cursor.

The Escape Codes

The LaserJet IID Technical Reference Manual lists 66 PCL escape sequence commands as shown in Figure 1, and require specific arguments. I will only cover about half of them in this series, and strongly recommend that you obtain the reference manual if you intend to do much PCL programming.

In this series, I am using the symbol <ESC> to indicate where the 1B Hex escape code will appear, since the escape code is not a printable character. The PCL codes are case sensitive—use lower and upper case as shown.

There are a few two-character escape sequences (e.g. <ESC>E and <ESC>9), but most of the commands are parameterized. The form is: 1) the ASCII 1B hex escape character, 2) a parameterized character in the range 21 to 2F Hex which indicates that the escape sequence is parameterized, 3) a group character in the range 60 to 7E Hex which specifies the group type of control being performed, 4) a value field with numeric characters in the range 30 to 39 Hex. It may be preceded by a "+" or "-" sign and may contain digits after a decimal point. If an escape sequence requires a value field and a value is not specified, a value of 0 is assumed, 5) a parameter character which specifies the parameter to which the previous value field applies. This is upper case except in a combined sequence.

Escape sequences can be combined if the first two characters (the parameterized character and the group character) are the same. An uppercase parameter character indicates the end of the sequence, while a lower case parameter character indicates that the sequence continues in a combined sequence. For example, the code <ESC>&l2A specifies that the page size is 8.5 x 11, and <ESC>&l0O specifies that the orientation is portrait. Since the first two characters after the <ESC> are the same, these can be combined as <ESC>&l2a0O. Note that the "A" in the first sequence is changed to lower case. I will try to avoid using combined sequences so that each sequence can be commented.The demonstration programs in this issue will use the following commands:

-<ESC>E - reset. The Reset returns the printer to the user default environment. It deletes any macros and temporary soft fonts. Any partial pages of data which have been received will be printed. H-P strongly recommends the use of <ESC> E at the beginning and end of each job.

-<ESC>*c#D - font ID. The font ID command is used to specify an ID number for use in subsequent font management commands. The ID number can range from 0 to 32767.

-<ESC>*c#F - font control. The Font Control command provides mechanisms for manipulating soft fonts with the # in the value field replaced with values as shown below:

0 - Delete all soft fonts
1 - Delete all temporary soft fonts
2 - Delete soft font (last ID specified)
3 - Delete Character Code (last ID and Character Code specified)
4 - Make soft font temporary (last ID specified)
5 - Make soft font permanent (last ID specified)
6 - Copy/Assign current invoked font as temporary (last ID specified)

-<ESC>(#X - primary font selection by ID

-<ESC>)#X - secondary font selection by ID. The printer maintains two independent font characteristic tables for use in selecting a primary font and a secondary font. You can select either font (only one is used at a time) with the SI (0F Hex) Shift In which selects the primary font, or SO (0E Hex) Shift Out control codes. Soft fonts can be assigned as the primary or secondary fonts by using their associated ID numbers.

-<ESC>*p#X - horizontal cursor positioning (dots). The Horizontal Cursor Positioning Command moves the cursor to a new position. If the value is preceded by a plus sign (+) it indicates that the new position is to the right and relative to the current position. If the value is preceded by a minus sign (-) it indicates that the new position is to the left and relative to the current position. No sign indicates that the value is an absolute distance which is referenced from the left edge of the logical page. If the indicated position is outside the logical page, the cursor is moved to the logical page limit.

-<ESC>*p#Y - vertical cursor positioning (dots). The Vertical Cursor Positioning Command moves the cursor to a new position. If the value is preceded by a plus sign (+) it indicates that the new position is downward and relative to the current position. If the value is preceded by a minus sign (-) it indicates that the new position is upward and relative to the current position. No sign indicates that the value is an absolute distance which is referenced from the top margin. If the indicated position is outside the logical page, the cursor is moved to the logical page limit.

Demonstration Program

I believe in starting with the simplest possible examples when learning a new technique. Something like C's "Hello World" example. The above escape sequences are enough to write a very simple demonstration program.

The example in Listing 1 resets the printer to the user default environment, positions the cursor 2 inches to the right of the left edge of the logical page and 2 inches down from the top margin, prints "TCJ", and ejects the page. Since no font selections were made, the user default font selected by the front control panel is used.

The example in Listing 2 is in two parts. First the printer is reset and a font ID number is assigned. Then an existing soft font is downloaded to the printer using the MS-DOS COPY command (be sure to use the /B binary switch). Then this font is made permanent so that it will not be deleted during a reset, the font is designated as the primary font, the cursor is positioned, and "TCJ" is printed. Finally, as an example of relative positioning, the cursor is positioned for a new line and is shifted vertically between the characters. The results are shown in Figure 2.

Different type faces and/or sizes can be selected for each character, and the characters can be positioned to overlap. This technique can be used to create special symbols or logos with only a few bytes of code, assuming that the fonts will be in the printer anyway. If the logo requires characters which are not in the fonts normally used, the characters can either be converted to a raster graphic image or a special font with only the required characters can be generated.

If you are using the Digi-Fonts font management program (see review in issue

Hewlett Packard Printer Control Language Commands
From the LaserJet IID Technical Reference Manual

<ESC>E - reset
<ESC>9 - clear horizontal margins
<ESC>= - half-line feed
<ESC>Y - display functions, enable
<ESC>Z - display functions, disable
<ESC>&a#C - horizontal cursor positioning (columns)
<ESC>&a#H - horizontal cursor positioning (decipoints)
<ESC>&a#L - left margin
<ESC>&a#M - right margin
<ESC>&a#R - vertical cursor positioning (rows)
<ESC>&a#V - vertical cursor positioning (decipoints)
<ESC>&d#D - underline enable
<ESC>&d@ - underline disable
<ESC>&f#X - macro control
<ESC>&f#Y - marco ID (assign)
<ESC>&k#G - line termination
<ESC>&k#H - horizontal motion index
<ESC>&l#A - page size
<ESC>&l#C - vertical motion index
<ESC>&l#D - line spacing
<ESC>&l#E - top margin
<ESC>&l#F - text length
<ESC>&l#H - paper source
<ESC>&l#L - perforation skip
<ESC>&l#O - orientation
<ESC>&l#P - page length
<ESC>&l#S - simplex/duplex command
<ESC>&l#U - left offset registration command
<ESC>&l#Z - top offset registration command
<ESC>&l#X - number of copies
<ESC>&p#X - transparent mode
<ESC> - end-of-line wrap
<ESC>(#B - stroke weight
<ESC>(#X - primary font selection by ID #
<ESC>(3@ - font default
<ESC>(ID - primary symbol set
<ESC>(s#H - primary pitch
<ESC>(s#P - primary spacing
<ESC>(s#S - primary style
<ESC>(s#T - typeface
<ESC>(s#V - primary height
<ESC>(s#W - character descriptor/data
<ESC>)#X - secondary font selection by ID #
<ESC>)3@ - font default
<ESC>)ID - secondary symbol set
<ESC>)s#B - secondary stroke weight
<ESC>)s#H - secondary pitch
<ESC>)s#S - secondary style
<ESC>)s#V - secondary height
<ESC>s#W - font descriptor
<ESC>)s#P - secondary spacing
<ESC>*b#W - transfer raster data
<ESC>*c#P - fill rectangular area
<ESC>*c#A - horizontal rectangle size (dots)
<ESC>*c#B - vertical rectangle size (dots)
<ESC>*c#D - font ID (specify)
<ESC>*c#E - character code
<ESC>*c#F - font control
<ESC>*c#G - area fill ID
<ESC>*c#H - horizontal rectangle size (decipoints)
<ESC>*c#V - vertical rectangle size (decipoints)
<ESC>*p#X - horizontal cursor positioning (dots)
<ESC>*p#Y - vertical cursor positioning (dots)
<ESC>*r#A - start raster graphics
<ESC>*r#F - raster graphics presentation
<ESC>*rB - end raster graphics
<ESC>*t#R - raster graphics resolution

Figure 1: PCL Escape codes.

```
<ESC>E
<ESC>*p600X
<ESC>*p600YTCJ
<ESC>E
```

Listing 1: Program to set "TCJ" using the default font.

TCJ

TCJ

Figure 2: Output from Listing 2.

First reset the printer and assign a font ID number.:

```
<ESC>E
<ESC>*c3D
```

Next, download the soft font:

```
COPY /B  fontfile.ext > PRN
```

Then, the font is made permanent, selected, and used:

```
<ESC>*c5F
<ESC>(3X
<ESC>*p600X
<ESC>*p600YTCJ
<ESC>*p600X
<ESC>*p740YT
<ESC>*p-30YC
<ESC>*p+30YJ
<ESC>E
```

Listing 2: downloading and using a soft font.

#39), the font can be generated, assigned a font ID number, designated as permanent, and loaded to the printer directly from DFI.

In the Future

Next time, we will take a close look at the structure of the H-P soft font files. We'll develop a program to download and manage the softfonts, and start dissecting the files in preparation for developing a program to read the height and width values.

After we can manage the font loading, and can obtain the height and width values, we'll develop a runoff program which will position and justify proportionally spaced fonts.

Another project will be to output pages in non-sequential order for special binding requirements. I am using PageMaker v3.0 for a 5.5 x 8.5 inch book which will be perfect bound. The camera ready copy will be run on 8.5 x 11 inch in landscape mode, and I need to place pages 24 and 1 on one side of the first sheet and pages 2 and 23 on the other side of that sheet, etc. I also need to emulate a duplex laser printer by running all the odd number pages first, then putting the paper back in and running all the even number pages on the other side of the sheets.

So far only the expensive ($3,500 and up) programs have provisions for imposition (non-sequential page output). Since my needs are to do this with PageMaker, I'll print to disk and then break it up into separate pages for outputting.

Remember, we are interested in your questions, tips, and problems. Take time to write or call. ●

(Continued from page 48)

ports). The I/O ports are just addresses in the memory map, mostly the first 32 locations. Most importantly is the assembler was free from the Motorola BBS. I also got a copy in the 68HC705 training manual. A large number of program samples are also available to show you how to program for things like 16 bit math.

There is most likely lots more to explain about the devices, but the best way is to use one. I had good feelings about the device to start with and it only got better. Code generation went very fast and was extremely easy. Motorola makes a whole line of simulators, all about $500 each. Most allow using RAM to store and test your program. We choose not to buy one and will find out later if we really needed it or not. I have written my code in such a way as to be able to hopefully test the unit even with some code problems. It is this modularity that I hope keeps me from needing an emulator.

Contest Anyone?

This reminds me of last times request for a contest. I mentioned it to Art, (our editor) and he seemed very interested. We are waiting to find out about your interest as well. Now I am not saying this project should be anything big or elaborate, much the opposite to be sure. The idea is to have fun! Create something beautifully useless, but educational all the same. It reminds me of the recent school contests called " challenges of the mind." The object is for students to create interesting working displays that perform a number of actions. We need to do the same.

One possible rule is that it must be self-contained. How about talk to you either visually or audibly. Perform at least 5 separate and different operations. Have at least 10 inputs, either as keypads or sensors. All with a limit of 5 to 10 IC devices, preferably the smaller the better. All we need now is your input.

ORCAD Latest

Went around with the PC Board house last week, seems ORCAD drill information is a bit off. The board house's program needed leading zeros and ORCAD liked to chop them off. When we got the board back and started checking hole sizes, many were way off. Their feed thrus are too large for the pads. The IC sockets are for smaller chips and sockets. They will not allow the larger pinned ZIF or LIF sockets to work. The holes for large transistors and TO5 regulators are many sizes too little.

All is not a loss as we only had two sets of boards made. What I can do is just change the tool assignments. The drill tape is a list of board locations and a tool number for drilling the hole. The numbers relates to a certain size hole drill. In many cases I need only change the assignment, but in others I need to assign different sizes to different parts of the device. My biggest concern is the pad left after some of the drilling operations. The pads consistently were under sized, and leave far too little for good soldering. I have already had a number of traces lift off the board after soldering. So little is left that minor heating will break the bond between the PCB and the trace.

My advice is to check and recheck both your choice of components as well as the drill information. I assumed that ORCAD did it correctly, but alas I was wrong. I had looked everything over several times, but had not put as much emphasis on hole size as I should have. Looking back I know where I was having problems and that was parts orders. I have had considerable problems with suppliers and find it has distracted me from doing the PCB work.

If I were doing this project over, I would design and prototype the entire board before ordering parts in quantity. We have a local used electronics place and I would have gotten all my parts there. Then breeze through the design stage and get one running. At this point put the project aside and start your purchasing. You know the design and concept work, now the question becomes finding the parts in quantity to meet your needs. Some changes in availability and thus changes in design will take place, but if your design is somewhat fixed, you will know the limits.

A typical problems has to do with some terminal strips I designed in. They are a common screw down strip with vertical mounting. I received 10 of the upright and 40 of the 45 degree style. The 45 degree made access to both the screw and wire holder easier. My supplier was short the one I needed (80 in all) but could get the 45 degree version faster. I opted to change to 45 degree design. Well when I got my boards and mounted things I discovered the height of the EMI FIlter blocked access to the screw holes. proto-typing means not only the design but the construction as well. Sometimes it is impossible to see all the relationships of parts from fact sheets, only actually building it points out the problems.

Next Time

Well this is about all this time. Promised myself and Art to get this to him a bit early. I hope to hear more about our contest from you and Art, so drop us a card if you are interested. I find sitting on my tractor working, or fishing, excellent places to think about totally useless ideas. Sounds like a great place to design that perfectly useless computer project to me. •

MOVING?

Make certain that TCJ follows you to your new address. Send both old and new address along with your expiration number that appears on your mailing label to:

THE COMPUTER JOURNAL
190 Sullivan Crossroad
Columbia Falls, MT 59912

If you move and don't notify us, TCJ is not responsible for copies you miss. Please allow six weeks notice. Thanks.

Registered Trademarks

It is easy to get in the habit of using company trademarks as generic terms, but these registered trademarks are the property of the respective companies. It is important to acknowledge these trademarks as their property to avoid their losing the rights and the term becoming public property. The following frequently used marks are acknowledged, and we apologize for any we have overlooked.

Apple II, II+, IIc, IIe, Lisa, Macintosch, DOS 3.3, ProDos; Apple Computer Company. CP/M, DDT, ASM, STAT, PIP; Digital Research. DateStamper, BackGrounder ii, DosDisk; Plu*Perfect Systems; Clipper, Nantucket; Nantucket, Inc. dBase, dBase II, dBase III, dBase III Plus; Ashton-Tate, Inc. MBASIC, MS-DOS; Microsoft. WordStar; MicroPro International Corp. IBM-PC, XT, and AT, PC-DOS; IBM Corporation. Z80, Z280; Zilog Corporation. Turbo Pascal, Turbo C; Borland International. HD64180; Hitachi America, Ltd. SB180 Micromint, Inc.

Where these, and other, terms are used in The Computer Journal, they are acknowledged to be the property of the respective companies even if not specifically acknowledged in each occurrence.

The Computer Corner

by Bill Kibler

It seems like just last week I wrote the last article, and yet here I am doing another one. That feeling of things moving too fast is no stranger around this place. Let's see if I can make some logical understanding of the last few weeks.

On Forth

I keep plugging away on little Forth projects. My latest one is writing a ROM based Forth system for embedded use. Needs to fit in 4K of memory with only 512 bytes of RAM available. It is a bit like the New Micros 68HC11 products but with more constraints. Hopefully next time a few more details on this one.

I ordered a few more publications on Forth and keep being amazed at just how many books are available. I can remember several years ago only having one or two to choose from. Now there is a rather broad selection available. The problem remains to find one that fits your needs. Unlike the Turbo series, there is usually only one book for a given topic, while Turbo may have ten or fifteen on a single aspect. I am not sure which is best, personally however I would like to read a few more opinions about a topic than just one.

One of my fellow workers asked me about Forth. Seems he had just read an article that pointed out how the new SUN SPARC work station has Forth in it. He has been wanting to buy one, since they are now under $10,000. They use Unix, but SUN had lots of problems configuring them, their answer was a Forth kernel under the Unix. They are not unlike many companies that use Forth for their diagnostics or startup test procedures. SPARC however is the first I have heard of that uses it for I/O. What was interesting is how this worker has suddenly found a interest in Forth. Link the words Forth and Unix and the whole world stops and listens.

6805

In the next few weeks my proto-types of the 68705 based lab system will be running. I finished all the software and should be testing it next week. I was concerned about the limited ROM space (3.7K), but it looks like I will have more than 1.5K left over. That is important, because I know the next version will have even more bells and whistles than this one. The 68705 however is so simple to work with, it takes very little code to achieve some powerful operations.

For those who use the Intel and not the Motorola line of products, I thought it might be helpful to review the 6805 style of devices. The 6800 series of devices have been around for a rather long time. I just tried to find out how long, but my books at hand miss saying when. I think (mostly guess) early 1970's as they are related to the 6502 (of Apple fame).

What Motorola did was create a linear style of CPU. I know the term, linear, is used often without much explanation. The explanation is simple, the same instruction can be used with every thing, or more simply—no special instructions! The internal design was so straight forward, that it could be built in discrete logic quite easily. I know that because I have worked on an industrial controller based on the 6800 design.

Internals

The basic concept is a register for doing the main work, and a index register for pointing into arrays. Now that is the 6805 design, which has been trimmed from the A and B registers of the 6800. One index not two are also features of the 6805. Right away some people wonder how you can program anything with so few registers. Simply use the entire address space as registers, that was the Motorola answer to more registers. The point to remember is, this is a processor for controlling simple tasks, not doing major feats of processing.

The hardware is simple, 4 ports of 8 bit data. No fancy handshakes for addressing other memory either. Just simple interrupts, a timer, and data ports. The 6805 can only do one thing, talk to I/O devices, typically switches, relays, and other on or off devices. There are a few variations on the main design, like a 68705, R3, P3, T3, and U3.

The 68705R3 is what I am using. It has 3 data ports and one analog port. The analog port has 4 channels which the system can read. The numbers 705 means it is an EPROM, or user programmable (and erasable) device. It also is very efficient by having a boot strap loader to be able to program its own memory. I got the bare programmer's board free from Motorola. About twenty dollars and a few minutes later I can program the 68705 from a ROM I programmed in a regular EPROM programmer. The boot strap loader signals the user when the chip has been programmed (copies the EPROM) and then signals when a verify has been completed successfully.

The newer HC series of the 6805 has a serial port instead of analog port. They also can be programmed by down loading your program over the serial port. This opens some interesting possibilities. I would like to see a version with both a serial and analog section however. I can always use bus switching to increase the number of lines in and out, where as adding an analog device is usually more costly and programming intense.

There are lots of new analog devices and I guess I could cover them next time. The HC devices were intended to use serial interfaced analog converters. The software interface really is not that much more than an internal port. Possibilities of more than 8 bits are also available when going serial. Their cost is now often below $10 and so the cost of extra board space can be less important. Overall my objections to NOT having a analog section in the HC series is rather mute at best.

Software

A quick review of software features shows that they have a number of nice options. You can index into tables using the X register as long as you don't exceed 256 bytes. This forced me to break a table into several sections before indexing. Your registers are all 8 bit but you have several memory instructions that work with more than 8. Most computers require all action to be on an internal register. The 6805 however lets you use memory much like a register. You can compare A register to memory locations. Clear memory, increase or decrease memory, ADD or SUB memory, logical operations on memory, as well as set and clear bits in memory (and I/O

(Continued on page 47)